Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

1. Einleitung



Quellenhinweis: Diese Folien entstanden in enger Kooperation mit Prof. R. Hahn.
Einige Folien stammen aus Präsentationen von Prof. G. Raffius, Prof. B. Humm und anderen Professoren des FBI

Eine Anekdote

Spitzenleute werden auf ein teures Seminar geschickt.
Sie sollen lernen, auch in ungewohnten Situationen Lösungen zu erarbeiten.
Am zweiten Tag wird einer Gruppe von Managern die Aufgabe gestellt, die Höhe einer Fahnenstange zu messen.
Sie beschaffen sich also eine Leiter und ein Maßband. Die Leiter ist aber zu kurz, also holen sie einen Tisch, auf den sie die Leiter stellen. Es reicht immer noch nicht. Sie stellen einen Stuhl auf den Tisch, aber immer wieder fällt der Aufbau um.
Alle reden durcheinander, jeder hat andere Vorschläge zur Lösung des Problems.
Eine Frau kommt vorbei, sieht sich das Treiben an. Dann zieht sie die Fahnenstange aus dem Boden, legt sie auf die Erde, nimmt das Maßband, misst die Stange, schreibt das Ergebnis auf einen Zettel und drückt ihn einem der Männer in die Hand.
Kaum ist sie um die Ecke, sagt einer: "Typisch Frau! Wir benötigen die Höhe der Stange und sie misst die Länge! Deshalb lassen wir Weiber auch nicht in den Vorstand.„

Was wird hier falsch gemacht?



Aussagen im Umfeld des Software Engineering

- „Programmieren klappt doch auch so!“
- „Wen interessieren schon Boxen und Linien?“
- „Ihr werdet nicht für bunte Folien mit Kästchen bezahlt!“
- „Prozesse haben noch nie ein Produkt geliefert!“

⇨ **Wozu brauchen Sie Software Engineering?**



Szenario



- Stellen Sie sich bitte folgendes vor:
  - ⇨ Sie arbeiten in einem Software-Projekt mit einem <u>Entwicklungsaufwand</u> von ca. <u>15 Mio. Euro</u> (<u>50 Personen über 3 Jahre</u>)
  - ⇨ Geplanter <u>Umsatz</u> des Projekts sind ca. <u>500 Mio. Euro</u>
  - ⇨ Sie müssen nach der Auslieferung <u>10 Jahre Wartung</u> bieten
  - ⇨ Die <u>Zukunft</u> <u>Ihrer Firma,</u> <u>Ihr Gehalt</u> und <u>Ihre Karriere</u> hängen am Erfolg des Projektes

Welche Probleme können dann auf Sie zukommen?

⇨ ... in der Entwicklung ?

⇨ ... im Management ?

Gründe für den Einsatz von Software Engineering

Komplexität:	Strukturierung, Verbergen von Details
Funktion:	Erfassen der Wünsche von Kunde (und eigener Firma)
Qualität:	Sicherheit, Regressanspruch, Zugänglichkeit
Wartung:	Gewährleistung, Dokumentation
Planung:	Umfang, Personal, Kosten, Termine, Qualität, Reporting
Organisation:	Teamgröße, Standortverteilung, unterschiedliche Sprache
Methodik:	Arbeitstechniken, Prozesse, Standards
…	

⇨ große Softwaresysteme in vorgegebener Qualität im vorgegebenen Zeit-Rahmen erstellen!

⇨ Vermittlung von Modellen, Methoden und Techniken für die Arbeit in solchen Projekten



Konkrete Gründe für die Notwendigkeit von Software Engineering

- In komplexen Projekten…
  - ⇨ kann (und will) eine Person nicht mehr alles wissen
  - ⇨ ist Aufgabenteilung erforderlich
  - ⇨ gibt es oft verteilte Entwicklung
  - ⇨ gibt es Wartungsverpflichtungen
  - ⇨ muss ein Budget realistisch abgeschätzt und dann eingehalten werden
  - ⇨ gibt es Mitarbeiterfluktuation während der Laufzeit
  - ⇨ gibt es Qualitätsauflagen (Intern oder vom Auftraggeber)
  - ⇨ …
- Software Engineering (oder auch "Software-Technik")…
  - ⇨ bietet Methoden um damit umzugehen!
  - ⇨ ist der einzige bekannte Lösungsansatz für komplexe Projekte!
  - ⇨ wird in der industriellen SW-Entwicklung von Informatikern erwartet!



SWE im Projekt (Auszug)

Kunde (Auftraggeber)

Termine
Kosten
Produkte

Anforderungen
Qualitäten I

Marktanforderungen

Geschäftsziele

Marketing

Manager (Controlling, Projektleitung)

Status

Anforderungen
Qualitäten II

Planung
Status

Qualitätssicherung

Durchführung von Maßnahmen

SW-Entwicklung

SWE im Projekt (Auszug)

Kunde (Auftraggeber)

UML

Termine
Kosten
Produkte

Anforderungen
Qualitäten I

Vorgehen
Prozesse
Planung

Marktanforderungen

Geschäftsziele

Marketing

Manager (Controlling, Projektleitung)

Status

Qualität
Prozesse

Qualitätssicherung

Anforderungen
Qualitäten II

Planung
Status

UML

SW-Entwicklung

Durchführung von Maßnahmen

OO, UML
Prozesse



Konzept der Veranstaltungen OOAD und SWE

- OOAD
  - ⇨ Objektorientierung
  - ⇨ Analyse und Design
  - ⇨ Dokumentation der Arbeit
    - UML
  - ⇨ "Handwerkszeug" für SW-Entwickler
    - Gutes Design
- SWE
  - ⇨ Gesamtkonzept / Zusammenhänge
    - Arbeit in großen Projekten
    - Anwendung des Handwerkszeugs
    - Qualitätssicherung
    - Organisation
- Praktikum OOAD
  - ⇨ Umgang mit dem Handwerkszeug
    - Erste Modellierung
    - "Entwickeln" eines Mini-Projekts
    - Arbeit mit der UML
    - Kennenlernen des CASE-Tools
    - Kennenlernen der Entw.umgebung (siehe Prakt.-Aufgabe letztes Semester)
- Praktikum SWE
  - ⇨ Anwendung der Verfahren aus OOAD und SWE
  - ⇨ Entwicklung eines Systems in kleinen Teams
  - ⇨ Möglichst hohe Realitätsnähe
    - Eigenständige Lösung der Aufgaben
    - Überprüfung durch Reviews
    - Aufteilung der Arbeiten im Team



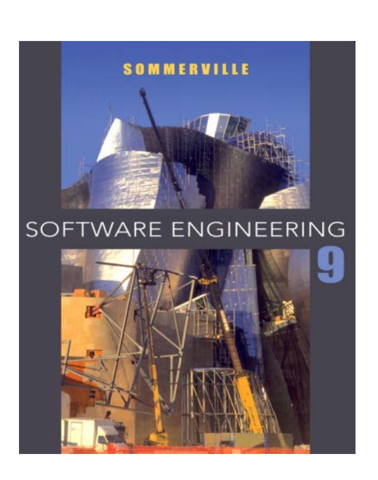
Literatur



"Methodische objektorientierte Software-Entwicklung"; Mario Winter; dpunkt-Verlag; 2005



"Software Engineering" Jochen Ludewig, Horst Lichter; dpunkt Verlag; 2006



"Software Engineering" Ian Sommerville; Pearson Studium; 9. Auflage, 2010



"Softwareentwicklung von Kopf bis Fuß" Dan Pilone, Russ Miles; O'Reilly Verlag; 2008



Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

2. Grundlagen des Software Engineering



Lernziele

- Sie sollen in diesem Kapitel verstehen,
  - ⇨ was alles zu Software gehört
  - ⇨ wie Software Engineering entstanden ist
  - ⇨ was Qualität von Software ausmacht
  - ⇨ wie wichtig die Qualitätseigenschaften für die Software sind
  - ⇨ welche Phasen in der SW-Entwicklung vorkommen
  - ⇨ welche Inhalte die Entwicklungsphasen haben

Hinterher ahnen Sie, warum in großen Projekten anders gearbeitet wird, als in kleinen Projekten!



Was ist das Problem?

- Komplexe SW-Projekte sind gekennzeichnet durch
  - ⇨ Software wird in mehreren Versionen weiterentwickelt und erweitert
    - Bedeutung von Dokumentation und Modellen für die Wartbarkeit
  - ⇨ Lange Entwicklungszeit
    - Bedeutung von Projektplanung und Terminüberwachung
  - ⇨ Einsatz von Entwicklerteams evtl. an mehreren Standorten
    - Bedarf an Standards und Regeln für Kommunikation und Dokumentation
  - ⇨ Lange Lebensdauer mit sich verändernden Anforderungen
    - Bedeutung von Dokumentation und Modellen für die Änderbarkeit

> ⇨ Die Methode VHIT (Vom Hirn Ins Terminal) ist zur Entwicklung solcher Projekte nicht geeignet
>
> ⇨ Wir benötigen andere Vorgehensweisen als für das Programmieren einer kleinen Anwendung
>
> ⇨ Wie kann man "ingenieurmäßig" Software entwickeln?



Begriffsklärung: Was ist Software?

- Software
  - ⇨ Enges Verständnis:
    ... unter Software subsumiert man alle immateriellen Teile, d.h. alle auf einer Datenverarbeitungsanlage einsetzbaren Programme
    (Lexikon der Informatik und Datenverarbeitung. H.-J. Schneider (Hrsg.), 1986)
  - ⇨ Mittleres Verständnis:
    Software (engl., eigtl. »weiche Ware«), Abk. SW, Sammelbezeichnung für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation (Brockhaus Enzyklopädie)
  - ⇨ Weites Verständnis = Unser Verständnis:
    Software: Menge von Programmen oder Daten zusammen mit begleitenden Dokumenten, die für ihre Anwendung notwendig oder hilfreich sind
    (Ein Begriffssystem für die Softwaretechnik, W. Hesse et al., 1984)

    Software: Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system
    (IEEE Standard Glossary of Software Engineering Terminology, ANSI 1983).



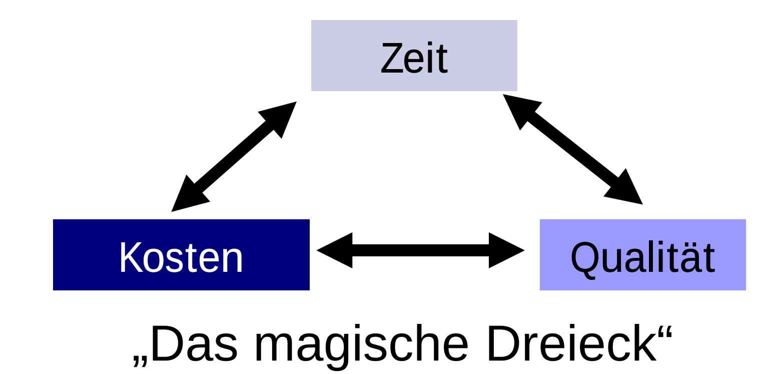
Was ist Software Engineering? (1)

- *Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen.*

(in Anlehnung an Balzert, SWT-Lehrbuch)



Was ist Software Engineering? (2)



- *Die Kunst, wie man Softwaresysteme in der erforderlichen Qualität, zum gewünschten Termin mit gegebenen Kosten erstellt*

⇨ d.h. Vermittlung von Modellen, Methoden und Techniken für:

- die konstruktiven Phasen der Software Entwicklung (Analyse, Entwurf, Implementierung), die verwendet werden um die Arbeitsergebnisse dieser Phasen zu strukturieren.
- Definition von Qualität (Qualitätseigenschaften, Qualitätsmodelle).
- zeitlichen Strukturierung in verschiedene Entwicklungsphasen (Phasenmodelle).
- Überprüfung der Phasenergebnisse (Test, Verifikation, Qualitätssicherung).

⇨ In Software Engineering lernen Sie eine Menge von nützlichen Werkzeugen kennen. Welches Werkzeug für welches Projekt angebracht ist, müssen Sie aber immer noch selbst entscheiden!



Geschichtliche Entwicklung (1)

- In der „Steinzeit“ der Software Entwicklung, in den 50er und 60er Jahren:
  - ⇨ Hardware war sehr teuer und wenig leistungsfähig
- Wo waren Computer eingesetzt? Welche Tätigkeiten konnte man damals durch den Computer unterstützen?
  - ⇨ Software und Computer fast ausschließlich im naturwissenschaftlichen Bereich
  - ⇨ Die eingesetzten Programme dienten hauptsächlich der Lösung mathematischer Probleme
  - ⇨ Die relativ kleinen Programme waren sehr preisgünstig oder wurde von den Anwendern selbst geschrieben.



Geschichtliche Entwicklung (2)

- Durch sinkende Hardwarepreise:
  - ⇨ Computer wurden für eine größere Zielgruppe erschwinglich.
- Es veränderten sich die Anforderungen an die Software:
  - Software wurde für die verschiedensten Bereiche gebraucht.
  - Der Umfang der Software wurde immer größer.
  - Ihre Komplexität war mit den bis zu diesem Zeitpunkt bekannten Vorgehensweisen für das „Programmieren im Kleinen“ nicht mehr zu beherrschen.

  ⇨ fehlerhafte und unzuverlässige Programme

  ⇨ nicht termingerechte Fertigstellung

  ⇨ geplante Kosten wurden überschritten

⇨ Dieser Zustand der Software Industrie wurde charakterisiert mit dem Begriff

"*Softwarekrise*"



Geschichtliche Entwicklung (3)

- Reaktion darauf:
  - ⇨ strukturierte Programmiersprachen (keine GOTOs)
  - ⇨ Entwicklungsmethoden

- Der Begriff „Software Engineering“ wurde zuerst auf einer Software-Engineering-Konferenz der NATO 1968 in Garmisch geprägt.
  - ⇨ Diese Konferenz gilt als Geburtsstunde des Software Engineering



Geschichtliche Entwicklung (4)

1. Erster Ansatz zur Entwicklung übersichtlicher Programme (1968)
   - ⇨ "strukturierte Programmierung": Vermeidung von GOTO-Anweisungen etc. (Dijkstra)

2. Entwicklung von Software-Engineering-Prinzipien (1968 - 1974)
   - ⇨ Strukturiertere Entwicklung von Programmen:
     - strukturierte Programmierung
     - schrittweise Verfeinerung
     - Geheimnis-Prinzip
     - Programmmodularisierung
     - Software-Lifecycle
     - Entitiy-Relationship-Modell
     - Software-Ergonomie

Frage: Wissen Sie, was sich hinter den Begriffen verbirgt?



Geschichtliche Entwicklung (4)

1. Erster Ansatz zur Entwicklung übersichtlicher Programme (1968)
   - ⇨ "strukturierte Programmierung": Vermeidung von GOTO-Anweisungen etc. (Dijkstra)

2. Entwicklung von Software-Engineering-Prinzipien (1968 - 1974)
   - ⇨ Strukturiertere Entwicklung von Programmen:
     - strukturierte Programmierung (keine GOTOs, nur Verzweigung, Schleifen, Pozeduren),
     - schrittweise Verfeinerung (Hierarchie von Funktionen, Top-down-Entwurf),
     - Geheimnis-Prinzip (rufende Prozedur kennt nur Schnittstellen der aufgerufenen Proz.),
     - Programmmodularisierung (Aufteilen des Programms in Teile)
     - Software-Lifecycle (mehrere Phasen der SW-Entwicklung)
     - Entitiy-Relationship-Modell (konzeptionelles Datenmodell mit Datenobjekten und Bez.),
     - Software-Ergonomie (bedienerfreundliche Benutzerschnittstelle)

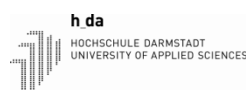


Geschichtliche Entwicklung (5)

3. Entwicklung von phasenspezifischen Software-Engineering-Methoden (1972 - 1975)
   - ⇨ Umsetzen der Software-Engineering-Prinzipien in Entwurfsmethoden:
     - Struktogramme
     - HIPO (Hirachical - Input - Process - Output)
     - Jackson, Constantine-Methode / Structured Analysis (Workflow), Structured Design (Spezielle Aufrufhierarchiediagramme)

4. Entwicklung von phasenspezifischen Werkzeugen (1975 - 1985)
   - ⇨ Einsatz von SE-Methoden mit maschineller Unterstützung z.B.
     - ProgrammInversion

       (zu einer Prozedur: Tabellarische Auflistung von gerufenen Prozeduren, rufenden Prozeduren, Verwendeten Bildschirmmasken, gelesene, geschriebene Dateien, ... – auch als Aufrufmatrix -> Kap. Metriken)
     - Batchwerkzeuge



Geschichtliche Entwicklung (6)

5. Entwicklung von phasenübergreifenden (integrierten) Software-Engineering-Methoden (ab 1980)

   ⇨ Automatische Weitergabe der Ergebnisse einer Phase des Software-Lifecycles an die nächste Phase: Methodenverbund

6. Entwicklung von phasenübergreifenden (integrierten) Werkzeuge (Case-Tools) (ab 1980)

   ⇨ Einsatz einer Datenbank als automatischer Schnittstelle zwischen den einzelnen Phasen des Software-Lifecycles.

Geschichtliche Entwicklung (7)

7. Definition verschiedener, konkurrierender objektorientierter Methoden (ab 1990)
   - ⇨ Es entstanden parallel verschiedene objektorientierte Analyse- und Entwurfsmethoden:
     - Booch
     - Jacobson
     - Rumbaugh
     - Shlaer/Mellor
     - Coad/Yourdon u. a.
   - ⇨ Die Methoden wurden in CASE Tools realisiert.

8. Integration der OO-Methoden zur UML- Unified Modeling Language (ab 1995)
   - ⇨ Jacobson, Booch und Rumbaugh schließen sich zusammen und entwickeln die UML. Die UML soll
     - die Schwächen der frühen OO-Methoden beseitigen und
     - ein weltweit gültiger, einheitlicher Standard werden



Geschichtliche Entwicklung (8)

9. Definition von mächtigen Methoden zur Entwicklung von Systemen (ab 1997)
   - ⇨ Es entstanden parallel verschiedene Vorgehensmodelle:
     - RUP (Rational Unified Process) seit 1998
     - V-Modell XT (seit 2005)
     - u. a.
   - ⇨ Die Methoden erfordern oft mächtige Tools.

10. Agile Software Entwicklung (seit 2000)
   - ⇨ Das „Agile Manifest" wird von Kent Beck und 16 weiteren anerkannten Größen der Software Entwicklung unterschrieben und veröffentlicht (2001)
   - ⇨ Gegenbewegung zu den „schwergewichtigen Methoden" als Rückbesinnung auf das eigentliche Ziel: lauffähige Software
   - ⇨ Es stellt Menschen, lauffähige Software, Zusammenarbeit und Änderungsbereitschaft über Prozesse, Dokumentation, Verträge und Pläne.



Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

2.1 Grundlagen des Software Engineering:

Softwarequalität



Ziel des Software Engineering

- Ziel:
  - ⇨ Softwaresysteme in der erforderlichen Qualität, zum gewünschten Termin mit gegebenem Kosten erstellen
- Welche Eigenschaften machen für SIE die Qualität einer Software aus?
  - ⇨ aus Entwicklersicht?
  - ⇨ aus Benutzersicht?
- Was ist Qualität / eine Qualitätseigenschaft?
  - ⇨ Und welche Qualität ist erforderlich? Wie definiere ich das?



Softwarequalitätseigenschaften

- Qualitätsmerkmale aus Benutzersicht
  - Funktionserfüllung
  - Effizienz
    - HW-Effizienz
    - SW-Effizienz (Performance)
  - Zuverlässigkeit
    - Robustheit
  - Benutzbarkeit
    - Robustheit
    - Fehlertoleranz
  - Sicherheit
- Qualitätsmerkmale aus Entwicklersicht
  - Erweiterbarkeit
    - Änderbarkeit
  - Wartbarkeit
    - Änderbarkeit
    - Verständlichkeit
    - Testbarkeit
  - Übertragbarkeit
  - Wiederverwendbarkeit



Softwarequalitätseigenschaften: Beispiele

- Funktionserfüllung

geplantes System  tatsächlich realisierter Funktionsumfang

- Effizienz
  - ⇨ Hohe Ablaufgeschwindigkeit (Performanz)
  - ⇨ wenig HW-Ressourcenverbrauch (z. B. geringer Speicherplatzbedarf)
    - HW wird immer billiger ⇒ verliert immer mehr an Bedeutung
    - Aber: immer noch wichtig bei hohen Stückzahlen z.B. bei embedded Systems
  - ⇨ Schnelle Algorithmen

Softwarequalitätseigenschaften: Bedeutung

- Die Umsetzung von Qualitätseigenschaften verursacht erhebliche Aufwände
  - ⇨ nachträgliche Änderungen von nicht eingeplanten Qualitätseigenschaften sind in der Regel sehr aufwändig
  - ⇨ Qualitätseigenschaften sind nicht unabhängig voneinander.
  - ⇨ Werden Maßnahmen zur Verbesserung einer Qualitätseigenschaft getroffen, so wirken sich diese möglicherweise auf andere Qualitätseigenschaften negativ aus.
- Beispiel:
  - ⇨ Wird die Effizienz verbessert, kann dies ein unzuverlässigeres System ergeben; die Robustheit kann sinken, das System kann schwerer änderbar sein (weil die Effizienzsteigerung z.B. durch Assemblereinsatz erreicht wurde)
  - ⇨ Hardwarenahe Programmiersprachen wirken sich natürlich auch auf die Wartbarkeit und Übertragbarkeit aus.

⇨ Die Festlegung der geforderten Qualitätseigenschaften ist genauso wichtig wie die Festlegung der Funktionalität



Maße für Qualität(seigenschaften)

- Bei der Spezifikation muss die Qualität der zu erstellenden Software definiert werden
  - ⇨ Es kann nach der Fertigstellung nachgeprüft werden, ob die Software den spezifizierten Qualitätsanforderungen genügt.
- Funktionserfüllung:
  - ⇨ Test: Funktioniert die Software so wie vereinbart?
  - ⇨ d.h. tut die Software das, was in der vom Auftraggeber abgezeichneten Spezifikation steht?
- Effizienz:
  - ⇨ spezifiziert durch Angabe von maximalen Antwortzeiten, Speicherverbrauch etc.
- Zuverlässigkeit:
  - ⇨ Spezifiziert durch Wahrscheinlichkeit, dass in der Zeit 0-t kein Fehler auftritt.
- Andere Maße, wie z.B. für Erweiterbarkeit oder Wartbarkeit:
  - ⇨ sind schwieriger zu definieren (⇨ Kapitel Softwaremetriken)

Qualität in der (deutschen) Sprache

- Die "Qualität", die ein Produkt hat, wird häufig in direkter Verbindung mit der Fehlerfreiheit gesehen
  - ⇨ Es gibt allerdings viele andere Qualitätseigenschaften, welche die Qualität eines Produkts bestimmen
  - ⇨ Die Zuverlässigkeit ist nur eine dieser Qualitätseigenschaften
  - ⇨ Die "erforderliche Qualität" hängt stark vom Projekt ab und wird auch oft von verschiedenen Beteiligten unterschiedlich gesehen
    - aus Sicht der Entwicklerfirma soll das Produkt z.B. erweiterbar sein und wenig Wartungskosten verursachen
    - aus Sicht des Anwenders soll es z.B. die gewünschten Funktionen bieten, zuverlässig und sicher sein
  - ⇨ Es gibt Qualität im Sinn von Zuverlässigkeit und Qualität als Oberbegriff für die gesammelten (Qualitäts)Eigenschaften eines Systems

⇨ Der Begriff „Qualität“ wird umgangssprachlich für beide Aspekte verwendet!



Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

2.2 Grundlagen des Software Engineering:

Phasen der Softwareentwicklung



Phasen der Softwareentwicklung

- Was denken Sie
  - ⇨ Welche Phasen gibt es im SW-Erstellungsprozess?
  - ⇨ Welche Phasen kennen Sie schon?



Der Software Lebenszyklus (Software Life Cycle)

Einführung, Betrieb

Analyse

Definition

Architektur

Design

Implementierung

Komponenten-Test

Integration

Integrations-Test

Abnahme

Phasen bei der Softwareentwicklung

- Im Praktikum "Programmieren"
  - ⇨ wurde immer eine Aufgabe gestellt, d.h. die Definition war vorgegeben
    - es wurde nie gefragt, ob die Entwicklung sinnvoll ist
  - ⇨ Sie mussten sich nur wenig Gedanken über den Aufbau des Systems machen
    - es ging direkt los mit der Implementierung – es war keine Planung erforderlich
- Bei der Entwicklung großer Systeme gibt es wichtige Phasen schon vor der Implementierung:
  - ⇨ Analyse:
    - Einarbeitung in die fachlichen Gegebenheiten
    - Soll ein System erstellt werden?
    - Ist eine Erstellung unter Kostengesichtspunkten ratsam und möglich?
  - ⇨ Definition (oft Teil der Analyse-Phase:
    - Spezifikation des Funktionsumfang und Qualität gemeinsam mit Benutzer / Auftraggeber
  - ⇨ Architektur und Entwurf:
    - Strukturierung des Systems und Entwurf der IT-Lösung



Analyse

- In der Analysephase
  - ⇨ wird erarbeitet (und verstanden) was die Software leisten soll
  - ⇨ Ein Systemanalytiker muss
    - das Anwendungsgebiet verstehen, den Ist-Ablauf aufnehmen,
    - begreifen was der Kunde will,
    - die Schwachstellen analysieren und Verbesserungsvorschläge prüfen,
    - zusätzliche Wünsche berücksichtigen.
    - **Verstehen, was die Software leisten soll**
  - ⇨ Ergebnis
    - Die fachlichen Anforderungen an die Software werden dokumentiert und mit dem Auftraggeber abgestimmt
    - Es wird analysiert ob die Aufgabe durchführbar (lösbar) ist
    - ob sie finanzierbar ist und ob das notwendige Personal vorhanden ist
    - Es wird eine erste Kostenkalkulation vorgenommen
  - ⇨ Fehler in der Analysephase
    - sind sehr teuer und wirken sich auf alle folgenden Phasen aus



Definition (oft auch Teil der Analysephase)

- In der Definitionsphase
  - ⇨ informellen Anforderungen der Analyse überführt in eine
  - ⇨ (möglichst) vollständige, konsistente Spezifikation

  - ⇨ Ergebnis
    - Beschreibt wie das Produkt aus Benutzersicht funktionieren soll
    - insbesondere wird der Abnahmetest mit dem Benutzer / Auftraggeber festgelegt
    - Soll normalerweise implementierungsunabhängig formuliert sein
    - Legt bereits Qualitätseigenschaften fest

  - ⇨ Die Definition beschreibt, „Was" das System tun soll



Architektur (Grobentwurf)

- In der Architektur
  - ⇨ wird die IT-Lösung auf einer hohen Abstraktionsebene entworfen
  - ⇨ es werden Festlegungen getroffen
    - Systembestandteile
    - Schnittstellen zu anderen Systemen
    - Interne Schnittstellen zwischen Systemteilen
    - Grundlegende Art der Umsetzung (z. B. Schichtenarchitektur)
  - ⇨ Ergebnis
    - Das System wird in grundlegende Bestandteile (Komponenten) zerlegt, die in Architekturdiagrammen (z.B. Komponentendiagramm) dargestellt werden
    - Tests für die grundlegenden Eigenschaften des Systems (Qualitätseigenschaften)
  - ⇨ Die Architektur beschreibt grob, „Wie“ das System aufgebaut ist



Design (Feinentwurf)

- Im Entwurf
  - ⇨ wird die IT-Lösung nahe am Zielsystem entworfen
  - ⇨ werden diverse Festlegungen getroffen
    - welches Betriebssystem,
    - welche Programmiersprache
    - welche Hardware,
    - Grundlegende Umsetzungsprinzipien (Design Patterns)
    - Datenbanksystem, notwendige Systemschnittstellen zu Fremdsystemen etc.
  - ⇨ Ergebnis
    - Das System (bzw. seine Komponenten) wird in Klassen zerlegt, die in Designdiagrammen (z.B. Klassen-, Sequenzdiagramm) dargestellt werden
    - Testpläne für die einzelnen Module und zusammengebaute Bestandteile
  - ⇨ Das Design beschreibt, „Wie" das System im Detail aufgebaut ist

> Bei kleinen Systemen werden Architektur und Entwurf oft zu einer Phase "Design" zusammen gelegt



Implementierung, Integration und Tests

- Implementierung und Test
  - ⇨ kennen Sie schon aus der Lehrveranstaltung Programmieren.
  - ⇨ Zumindest kleine Programme wurden von Ihnen implementiert und (grob) getestet
- Komponenetentest, Integration und Integrationstest bei großen Systemen
  - ⇨ Bei großen Systemen kann man das System nicht als Ganzes implementieren und testen
  - ⇨ implementiere und teste zuerst die einzelnen Module (=Funktionen bzw. Komponenten) (Funktionen- bzw. Komponententest)
  - ⇨ montiere anschließend diese getesteten Module schrittweise zusammen (Integration) und teste sie (Integrationstest, Systemtest).
  - ⇨ Vor Beginn der eigentlichen Codierung:
    - Es wird entschieden, in welcher Reihenfolge die Komponenten implementiert und zusammengebaut werden (z. B. Top-Down- oder Bottom-Up-Vorgehensweise).
    - Detaillierte Testpläne für die einzelnen Funktionen, Komponenten, für die Teilsysteme und das Gesamtsystem sind, falls nicht schon in vorhergehenden Phasen geschehen, zu erstellen.



Abnahme

- Abnahme
  - ⇨ Kennen Sie schon aus dem Programmierpraktikum.
  - ⇨ Es wird geprüft:
    - Stimmen die Leistungen des Systems mit den in der Spezifikation vereinbarten Leistungen überein? ("Abnahmetest")
    - Ein Abnahmetest für ein großes System muss systematisch geplant werden und ist wesentlich aufwendiger als im Programmierpraktikum
    - Ist ein externer Auftraggeber vorhanden, überprüft dieser die Systemfunktionen mit einem eigenen Prüfkatalog
  - ⇨ Ergebnis
    - Im Programmierpraktikum: Sie bekamen Ihre Abnahme – fertig –
    - Bei der kommerziellen Softwareentwicklung: Die Software kommt zum Einsatz d.h. der Betrieb des Systems



Einführung

- In der Einführung ("Rollout")
  - ⇨ wird das neue System in Betrieb genommen und evtl. ein altes System abgeschaltet
    - Evtl. stückweise Umstellung
    - Evtl. mit Möglichkeit zur Wiederherstellung des alten Systems ("Rollback")
    - Evtl. anfangs Parallelarbeit im alten und neuen System
  - ⇨ Tätigkeiten
    - Planung der Einführung (Termine, Ankündigungen, Verteilung von Anleitungen usw.)
    - Schulung des Personals
    - Aufbereitung/Migration von Daten, Erfassung von Daten
    - Installation der Software / der Hardware
    - Bereitstellung einer Hotline
    - Eigentliche Umstellung
  - ⇨ Ergebnis
    - Das System ist bereit für den "Normalbetrieb"

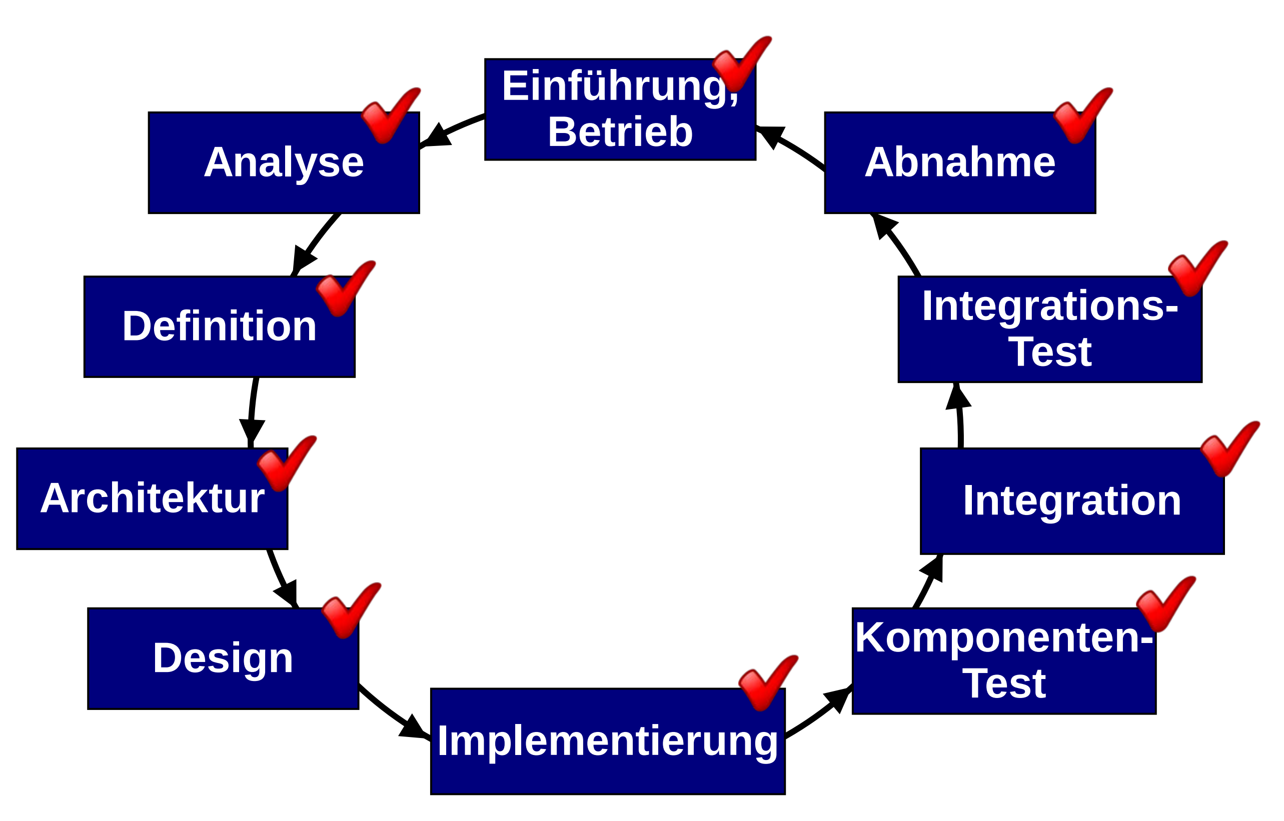
Betrieb

- Betriebsphase
  - ⇨ ist die längste aller Phasen
  - ⇨ verursacht oft sehr hohe Kosten

    Frage: Was meinen Sie? Wie lang? Wie viel % der Kosten?
  - ⇨ ca. 80% der Kosten aller Phasen
  - ⇨ d.h. es rentiert sich Aufwand zu investieren für die Erstellung
    - übersichtlich entworfener und
    - gut getesteter Programme
    - mit guter Dokumentation
  - ⇨ Ergebnis
    - Fehler und Unzulänglichkeiten am Produkt werden hier bereinigt.
    - SW wird an geänderte Einsatzzwecke und betriebliche Abläufe angepasst.
    - Oft ist das Zurückgehen über mehrere Phasen hinweg notwendig.

Auch wenn diese Phase einen Entwickler wenig interessiert – viele der Aktivitäten im Software Engineering sind nur durch die Aufgaben in der Betriebsphase zu begründen!

Der Software Lebenszyklus (Software Life Cycle)



- Die Detaillierung der einzelnen Phasen hängt vom Bedarf ab:
  - ⇨ Zusammenlegung „uninteressanter“ Phasen
  - ⇨ Verfeinerung „interessanter“ Phasen



Phasenmodelle

- Durch den Einsatz von Phasenmodellen wird ein Softwareprojekt zeitlich strukturiert. Es wird festgelegt:
  - ⇨ Welche Arbeiten sind in welchen Phasen zu erledigen?
  - ⇨ Welche Arbeitsergebnisse müssen in den Phasen erzeugt werden?
  - ⇨ Welche Personen haben die Ergebnisse zu erarbeiten?
  - ⇨ Wie sollen die Ergebnisse dokumentiert werden?
  - ⇨ Welche Termine müssen eingehalten werden?
  - ⇨ Welche Standards und Richtlinien sind einzuhalten?
  - ⇨ Phasenmodelle regeln auch die Zusammenarbeit zwischen den am Softwareerstellungsprozess Beteiligten.
  - ⇨ Es existieren verschiedene Phasenmodelle für die Softwareentwicklung.



Wasserfallmodell



Wasserfallmodell -> V-Modell (ursprüngliche Version)

(Einführung, Betrieb)

Entwicklung	Verbindung	Test
Anforderungs-definition	Anwendungsszenarien	Abnahmetest
Grob-entwurf	Testfälle	Systemtest
Fein-entwurf	Testfälle	Integrations-test
Modul-Imple-mentierung	Test-fälle	Modultest

Weitere Modelle im Kapitel 11: Vorgehens- und Prozessmodelle



Ergebnisse der Phasen

- Als Ergebnis jeder Phase entstehen Dokumente,
  - ⇨ die als Grundlage für das Weiterarbeiten dienen.
  - ⇨ **Fehler in frühen Phasen** verursachen (sehr) **hohe Kosten** in späteren Phasen z.B.: Auswirkungen von Fehlern oder missverständlichen Formulierungen in der Spezifikation, die erst zum Zeitpunkt der Abnahme festgestellt werden
    - alle Phasen müssen noch mal durchlaufen werden,
    - man muss sich in den Aufbau des Programms hineindenken und Änderungen im Code vornehmen
    - System ist neu zu testen (Komponenten- und Integrationstest des gesamten Systems)
    - Dokumentation ist anzupassen

- Qualitätssichernde Maßnahmen (z.B. Reviews und Codeinspektionen)
  - ⇨ prüfen die Phasen-Ergebnisse
  - ⇨ und evtl. auch innerhalb der Phasen - an so genannten Meilensteinen - anfallende Zwischenergebnisse

Eine ausführliche Betrachtung von Phasenmodellen (auch mit mehreren Zyklen) kommt später!



Kontrollfragen

- Gehören Dokumentation und Skizzen auch zur Software?
- Was war die Software-Krise?
- Was ist strukturierte Programmierung
- Was ist schrittweise Verfeinerung
- Was ist Qualität von Software?
- Was ist das "magische Dreieck"?
- Was passiert in der Phase "Betrieb"?
- Diskussion: Warum verkauft sich Software mit Fehlern trotzdem?

Jetzt ahnen Sie, warum in großen Projekten anders gearbeitet wird, als in kleinen Projekten!



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

3. Anforderungsanalyse



Häufige Mängel in Software-Projekten… sind vermeidbar

How the customer explained it

How the Project Leader understood it

How the Analyst designed it

How the Programmer wrote it

How the Business Consultant described it

How the project was documented

What operations installed

How the customer was billed

How it was supported

What the customer really needed



Phasenmodell: Zur Erinnerung - V-Modell

Anforderungs-definition ---Anwendungsszenarien---> **Abnahmetest**

Grob-entwurf ---Testfälle---> **Systemtest**

Fein-entwurf ---Testfälle---> **Integrations-test**

Modul-Imple-mentierung ---Test-fälle---> **Modultest**

Diese Phasen kommen auch in anderen Phasenmodellen vor...



Lernziel Anforderungsanalyse

- Sie sollen in diesem Kapitel verstehen,
  - ⇨ Was in der Anforderungsanalyse gemacht wird
  - ⇨ Warum die Anforderungsanalyse so schwierig ist
  - ⇨ Woher die Kommunikationsprobleme in der Anforderungsanalyse kommen
  - ⇨ Welche Arten von Anforderungen es gibt
  - ⇨ Welche Dokumente zu einer Anforderungs-Spezifikation gehören

Anschließend können Sie ein System so spezifizieren, dass es beauftragt oder entwickelt werden könnte



Typischer Beginn eines Projekts I

- Auftraggeber
  - hat ein Problem und sucht eine Lösung
  - kennt sich oft nicht mit SW aus
  - Formuliert (evtl.) die Randbedingungen
  - hat ein gewisses Budget
  - Formuliert den Bedarf
  - hat eine eigene Sprache
  - hat eigene Prozesse

- Auftragnehmer (SW-Firma)
  - soll einen Lösungskonzept erstellen
  - hat ein bestimmtes Fachwissen in SW
  - kennt die Randbedingungen nicht
  - soll ein Angebot machen
  - kennt sich mit der Anw.domäne nicht aus
  - hat eine eigene Sprache
  - hat eigene Prozesse

Lasten-heft

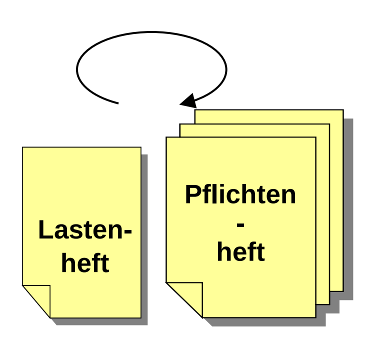




Typischer Beginn eines Projekts II



- Ziele
  - ⇨ Abgestimmte Beschreibung des Lieferumfangs
    - aber nur mit eingeschränktem Aufwand !
    - angestrebter Detaillierungsgrad ist definiert im Vorgehensmodell
  - ⇨ Vertragsinhalte festlegen
    - Funktionalität, Bezahlung, Termine, Abnahme, Strafen,…
    - Iteration des Pflichtenhefts (und evtl. Lastenhefts)
    - Erarbeiten eines gemeinsamen Verständnisses





Inhalt eines Lastenhefts (Idealergebnis)

Titel	Systemname, Datum, Autor
I. Zielbestimmung und Zielgruppen	Ausgangssituation, grundsätzl. Zielsetzung
I.a. Produktperspektive	Erwartete Verbesserung
I.b. Einsatzkontext	Einsatzmöglichkeiten
II. Funktionale Anforderungen	Welche Leistung liefert das Produkt?
II.a. Produktfunktionen	Funktionalität
II.b. Produktschnittstellen	Schnittstellen zu anderen Systemen
II.c. Anwenderprofile	potenzielle Benutzer
III. Nichtfunktionale Anforderungen	Wie (gut) wird die Leistung erbracht?
III.a. Qualitätsanforderungen	*siehe nächste Folie*
III.b. Technische Anforderungen	Einsatzumgebung
IV. Lieferumfang	Dokumente, Systeme, Code, Wartung
V. Abnahmekriterien	Minimalkriterien für die Akzeptanz
Anhänge	Sonstiges

Quelle: M. Winter "Methodische objektorientierte Softwareentwicklung"



Zur Erinnerung: Softwarequalitätseigenschaften

Die müssen erarbeitet werden!

- Qualitätsmerkmale aus Benutzersicht
  - Funktionserfüllung
  - Effizienz
    - HW-Effizienz
    - SW-Effizienz (Performance)
  - Zuverlässigkeit
    - Robustheit
  - Benutzbarkeit
    - Robustheit
    - Fehlertoleranz
  - Sicherheit
- Qualitätsmerkmale aus Entwicklersicht
  - Erweiterbarkeit
    - Änderbarkeit
  - Wartbarkeit
    - Änderbarkeit
    - Verständlichkeit
    - Testbarkeit
  - Übertragbarkeit
  - Wiederverwendbarkeit

Versteckte Anforderungen

- Nur wenige Menschen werden Ihnen sofort Anforderungen nennen
  - ⇨ Oft erhalten Sie Lösungsvorschläge
  - ⇨ es gibt meist viele versteckte Anforderungen
    - Ihre Aufgabe ist es, diese Anforderungen herauszufinden!
    - Es gibt oft versteckte Hinweise auf Anforderungen in Dokumenten und Präsentationen… schauen bzw. hören Sie genau hin!

Information	Hinweis z.B. auf
Einsatzort des Systems	Wartungsart: Remote oder vor Ort
Vertriebsweg und Markt	Art der Installation
Benutzungsfrequenz	Performanz, Hardware-Auslegung
Folgen bei Ausfall	Robustheit, Sicherheit
Anwenderprofil	Internationalisierung, Bedienbarkeit
Geschäftsplanung	Erweiterbarkeit
Probleme	kritische Anforderungen

Übung CocktailPro (I): Firmenpräsentation Cocktail4U

Dieses System werden Sie im Praktikum entwickeln!

- Cocktail4U GmbH
  - ⇨ 5 Mitarbeiter
  - ⇨ 100% Tochter der DfF GmbH (Dosierer für Fertigungsstraßen)
    - automatisierte Dosierer, Waagen, Mischer uvm.
    - Einsatz in Brauereien, Konditoreien, Lebensmittelverpackung
- Geschäftsidee Cocktail4U: CocktailPro
  - ⇨ Wandgerät für Kneipen und Bars
  - ⇨ hochwertige Cocktails auf Knopfdruck
- Situation:
  - ⇨ DfF hat die Dosierer, Waage, etc.
  - ⇨ es fehlt nur noch die Steuer- und Kontrolleinheit
  - ⇨ Cocktail4U beauftragt die Entwicklung ... bei IHNEN!?





Übung CocktailPro (II): Lastenheft

CocktailPro, Cocktail4U GmbH

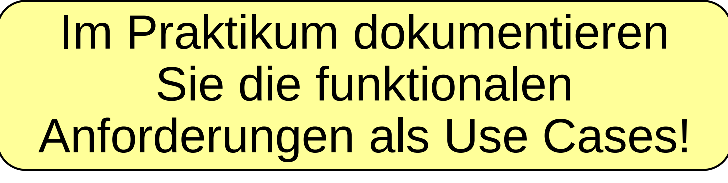
I.	Zielbestimmung und Zielgruppen	
	I.a. Produktperspektive:	Standardausstattung in der Gastronomie
	I.b. Einsatzkontext:	Cocktailbar, Restaurant, Kneipe
II.	Funktionale Anforderungen	
	II.a. Produktfunktionen:	Cocktail auf Knopfdruck
	II.b. Produktschnittstellen:	Wasser, Strom, Dosierer
	II.c. Anwenderprofile:	Wirte; geringe Ausbildung
III.	Nichtfunktionale Anforderungen	
	III.a. Qualitätsanforderungen:	einfaches GUI (Knopf = Cocktail), 1 Cocktail in 30 Sek
	III.b. Techn. Anforderungen:	Innenraum, Wandmontage
IV.	Lieferumfang:	Gesamtsystem, Code, Wartung
V.	Abnahmekriterien:	50 Cocktails ohne Störung
	Anhänge:	Systemskizze, Beschreibung von Dosierer, etc.



Übung: CocktailPro (III): Systemüberblick aus Sicht des Auftraggebers

Rezeptur Prozessor
Pitu
Zucker
Eis
Limette
Dosierer
Dosierer
Dosierer
Stück Dosierer
Stampfer
Mischer
Entleerer
Waage
Rezept
Schritt
Schritt

Übung: CocktailPro (IV): Offene Punkte





- Obwohl das Lastenheft alle Punkte anspricht, fehlen noch viele Infos!
- Unklar ist zum Beispiel:
  - ⇨ Wer liefert den Rechner, das Gehäuse?
  - ⇨ Woher kommen die Rezepte?
  - ⇨ Wie werden (neue) Rezepte eingespielt? (Remote? vor Ort? Schnittstelle?)
  - ⇨ Wie viele Rezepte werden angeboten? (Das GUI Konzept verlangt je 1 Knopf)
  - ⇨ GUI (Sprachen, Internationalisierung, Sonderzeichen, Auflösung, Farbe)
  - ⇨ Was passiert bei einer Störung? (Fernwartung oder Führung durch System)
  - ⇨ Wer zahlt bei Störungen in zugelieferten Teilen?
  - ⇨ Wie viele Zutaten können gemischt werden (# Dosierer)?
  - ⇨ Was wird gewartet? Ausgelieferte Systeme oder nur die Software? Rezepte?
  - ⇨ Gesetzliche Auflagen (z.B. Hygiene, Reinigungskonzept)
  - ⇨ Aussagen zu nicht-funktionalen Anforderungen (z.B. Erweiterbarkeit)
  - ⇨ Auflagen für Programmiersprache, Betriebssystem
  - ⇨ …



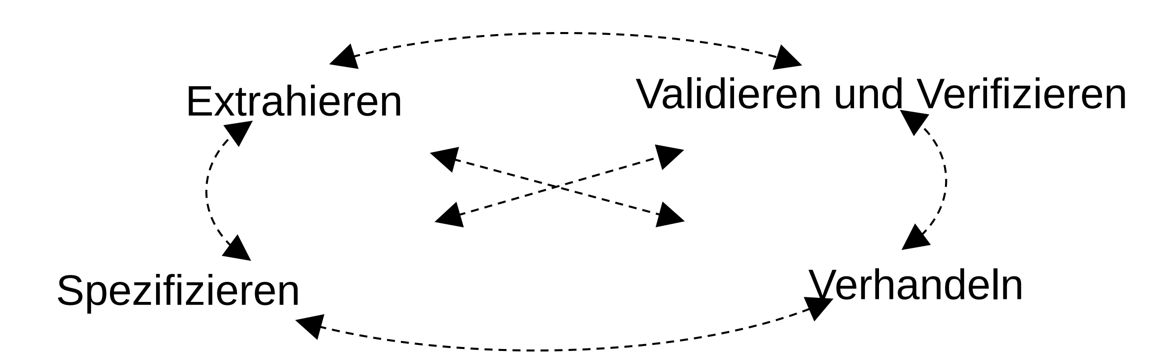
Erstellung eines Pflichtenhefts (I)

- Annahme: Es liegt ein "gutes" Lastenheft vor

- Was fehlt noch?
  - ⇨ gemeinsame Sprache
  - ⇨ gemeinsames Verständnis der Funktion
  - ⇨ gemeinsames Verständnis der Randbedingungen
  - ⇨ gemeinsame Vorstellung über das Zielsystem

Details ...
- ⇨ Glossar
- ⇨ Funkt. Anf.
- ⇨ n.-funkt. Anf.
- ⇨ GUI

- Wie kommt man dahin?



nach K. Pohl "Process-Centered Requiremens Engineering", 1996



Erstellung eines Pflichtenhefts (II)

Extrahieren
Validieren und Verifizieren
Spezifizieren
Verhandeln

- Extrahieren (Extraction / Elicitation)
  - Kennenlernen der Anwendungsdomäne, Aufnahme Istablauf
  - Gemeinsames Brainstorming, Diskussionen, Anwenderbefragungen, Dokumenten- und Literaturstudium
  - Schwachstellen, Verbesserungsvorschläge, Wünsche, verborgene Anforderungen finden
  - Lösungsalternativen finden und auswählen
- Spezifizieren (Specification)
  - Thema verständlich und kommunizierbar machen, dokumentieren
- Validieren
  - Werden Erwartungen der Benutzer / Auftraggeber wirklich erfüllt?
  - Diskussionen von Auftragnehmer & Auftraggeber
- Verifizieren
  - Konsistenz der Einzelspezifikationen prüfen, Technische Überprüfung z.B. durch Spezialisten, Test-Team
- Verhandeln (Negotiation)
  - Einigung auf Reihenfolge, Priorität, Umfang etc. einer Realisierung mit allen Stakeholdern



Bestandteile eines Pflichtenhefts

Alle Beschreibungen beziehen sich auf den Problemraum! Lösungen im Detail sind hier noch nicht von Interesse!

- Funktionale Anforderungen
  - konkrete Anwendungsfälle und Abläufe (z. B. Use Cases, Story Cards = Text)
  - Benutzer, Akteure
- Nicht-funktionale Anforderungen
  - Systematische Betrachtung der Qualitätseigenschaften
  - Priorisierung der Qualitätseigenschaften
- Benutzungsschnittstelle (GUI=Graphical User Interface / HMI=Human Machin Interface)
  - Skizzen und Snapshots, Navigationskonzepte, Dialogführung
- Glossar
  - Zentrale Sammlung der Definitionen aller wichtigen Dinge der Problemwelt
  - den Auftraggeber nicht mit neuen Begriffen "beglücken"!
  - Universalausdrücke vermeiden / genau spezifizieren: (System, Komponente, …)
- Sonstiges
  - z.B. Modellierung der Daten des interessierenden Realweltausschnitts (Analyse-Klassendiagr.)

Die konkrete Art der Dokumentation wird mit dem Geschäftspartner festgelegt!



Tätigkeiten bei der Anforderungsanalyse und Dokumentationsformen

- Definition der Funktionalität
  - ⇨ siehe OOAD: Use Cases, Aktivitätsdiagramme, Zustandsdiagramme



- Definition der nichtfunktionalen Anforderungen
  - ⇨ Qualitätseigenschaften als Text dokumentieren



- Erstellung von HMI-Skizzen und Navigation
  - ⇨ Vorlesung "Entwicklung nutzerorientierter Anwendungen" (ENA)



- Glossar
  - ⇨ Normale alphabetische Definitionsliste (evtl. mit Index)



- Sonstiges
  - ⇨ z.B. Modellierung der Anwendungsdomäne: siehe OOAD: Analyseklassenmodell



Anforderungsanalyse - Was ist eigentlich daran so schwer?

- Nette Gesprächsrunden (viel Kaffee, viele Kekse)…
  - ⇨ …gemeinsames Verständnis finden…
  - ⇨ …Aufschreiben der Ergebnisse…
  - ⇨ … und fertig???
- Nicht ganz, denn…
  - ⇨ … in der Regel können nicht alle Anforderungen umgesetzt werden
    - Priorisierung, Planung und Verhandlung ist erforderlich
    - Ablehnung von Anforderungen ist oft nötig
  - ⇨ … die Anforderungen ändern sich mit der Zeit
    - Umgang mit neuen / geänderten Anforderungen
    - Konsistenz der Anforderungen erhalten
    - Verwaltung der Anforderungen

⇨ Change Request Management
⇨ Requirements *Engineering* statt Requirements *Management*

Kontrollfragen zur Anforderungsanalyse

- Welches Ziel verfolgt die Anforderungsanalyse?
- Was wird in der Anforderungsanalyse gemacht?
- Warum ist die Anforderungsanalyse so schwierig?
- Welche Arten von Anforderungen gibt es?
- Welche Dokumente gehören zu einer Anforderungs-Spezifikation?
- Welche Methode eignet sich zur Dokumentation von funktionalen Anforderungen?
- Was geschieht, wenn man ein Projekt ohne aufwändige Anforderungsanalyse angeht / angehen muss?

Könnten Sie jetzt ein System so spezifizieren,
dass es beauftragt oder entwickelt werden könnte?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

4. Abnahmetest



V-Modell

Anforderungs-definition
Anwendungsszenarien
Abnahmetest
Grob-entwurf
Testfälle
Systemtest
Fein-entwurf
Testfälle
Integrations-test
Modul-Imple-mentierung
Test-fälle
Modultest



h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Lernziel Abnahmetest

- Sie sollen in diesem Kapitel verstehen,
  - Was ein Abnahmetest ist
  - Warum ein Abnahmetest wichtig ist
  - Wie man Testfälle für die Abnahme findet

Anschließend können Sie ausgehend von einer Spezifikation die Abnahmetests spezifizieren.



Was ist ein Abnahmetest und wie führe ich ihn durch

- Was ist ein Abnahmetest?
  - ⇨ Prüfung, ob erstelltes System = spezifiziertes System
  - ⇨ die Abnahmetests sind Bestandteil des Vertrags
  - ⇨ Nach erfolgreicher Abnahme wird bezahlt

- Wie führe ich einen Abnahmetest durch?
  - ⇨ Zusammen mit dem Auftraggeber
  - ⇨ Mit formeller Protokollierung

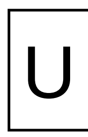
FURPS-Modell

- Was muss getestet werden?
  - ⇨ F: Functonality (Funktionalität)
  - ⇨ U: Usability (Benutzbarkeit)
  - ⇨ R: Reliability (Zuverlässigkeit)
  - ⇨ P: Performance (Leistung)
  - ⇨ S: Supportability (Wartbarkeit)

    ...
- Funktionstests
  - ⇨ Überprüfung auf korrekte Funktion des Systems

kommt gleich im Detail !

- Benutzbarkeitstests
  - ⇨ Überprüfung der Bedienoberfläche durch Endanwender
  - ⇨ Intuitive Bedienung, Lesbarkeit, Reaktivität, ...



Tests: (RPS)

- Zuverlässigkeitstests
  - ⇨ Zuverlässigkeit: Max. Anzahl der Abstürze/Fehlfunktionen in Zeitintervall T
  - ⇨ Beim Funktionstest bekommt man auch Hinweise auf die Zuverlässigkeit
  - ⇨ Eigentlicher Zuverlässigkeitstest erst im Einsatz
- Leistungstests
  - ⇨ Überprüfung der Nichtfunktionalen Anforderungen zu
    - Performance: Zeitvorgaben,
      d. h. Überprüfung der Schnelligkeit durch Überprüfung der Zeitvorgaben
    - Robustheit: Belastungs- oder Stresstest,
      d. h. Funktionstests auch bei maximaler Last, maximaler Durchsatz, hohem Datenaufkommen etc.
- Wartbarkeitstest
  - ⇨ Überprüfung von Code und Dokumentation auf Verständlichkeit
  - ⇨ z. B. durch Anwendung von Metriken (später in der Vorlesung)

R

P

S



Tests: Funktionstest im Detail

F

- Woher kennen wir die gewünschte Funktionalität?
  - ⇨ In der Anforderungsanalyse wurden die Funktionalität durch Eingaben und die darauf erwartete Reaktion des Systems beschrieben
- Annahme: Wir haben Anwendungsfälle beschrieben
  - ⇨ In Anwendungsfällen sind Eingaben ins System und Ausgaben vom System dokumentiert
  - ⇨ Beim Abnahmetest werden die einzelnen Anwendungsfälle ein- oder mehrmals durchgespielt
  - ⇨ Um nachzuprüfen, ob die - vom Anwendungsfall beschriebene - Funktionalität wie erwartet abläuft

Wie extrahiert man die Testfälle und die dazugehörenden Daten aus Anwendungsfällen?



Ableiten von Funktionstests (I) aus Use Cases

Trigger	**Login-System fragt Kunden nach PIN**		
	Step #	Action	
Basic Course	1	System fordert zur PIN-Eingabe auf	
	2	Kunde gibt PIN ein	
	3	System validiert PIN	
	4	System gibt Zugang frei	
Alternate Course	Step #	Condition	Action(s)
	*	Kunde bricht ab	Abbruch
	2a	Keine Eingabe für 30 Sek.	Abbruch
	3a	PIN stimmt nicht und Versuche <3	Goto 1
	3b	PIN stimmt nicht und Versuche>=3	Abbruch

Gefährlich! Wirklich immer?



Ableiten von Funktionstests (II): Essentielle Aktivitäten ohne Fehler- und Ausnahmesituationen

Zur Anschauung: Darstellung als Aktivitätendiagramm

Aufforderung PIN-Eingabe

Kundenreaktion

PIN-Validierung

System gibt Zugang frei



Beispiel: Ableiten von Funktionstests (III): Aktivitäten mit Ausnahmen

Aufforderung PIN-Eingabe

Kundenreaktion

keine Eingabe für 30 Sek.

Kunde bricht ab

PIN-Eingabe erfolgt

PIN falsch und
< 3 Fehlversuche

PIN-Validierung

PIN falsch und
>= 3 Fehlversuche

PIN als richtig
validiert

System gibt Zugang frei

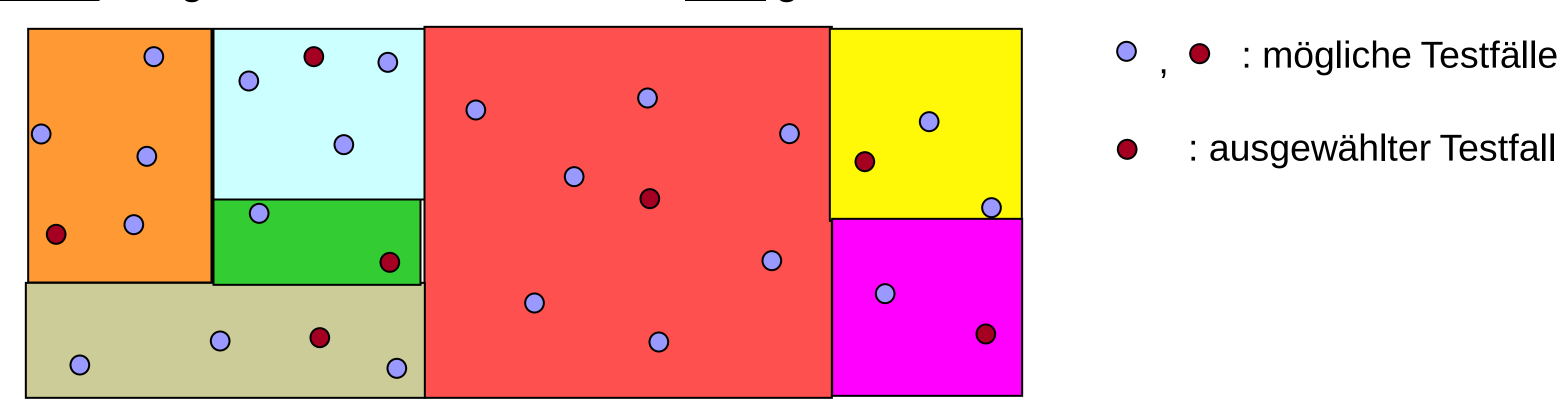
Abbruch

Ableiten von Funktionstests (IV)

- Man kann das System nicht mit allen möglichen Eingabedaten testen.

⇒ man teilt die Menge der Testfälle (der möglichen Eingaben) in Klassen ein, so dass man sagen kann:

  - Falls ein Testfall aus der Menge der Testfälle der Klasse richtig abläuft, so laufen auch die anderen mit großer Wahrscheinlichkeit richtig ab.



- Testplan besteht aus der Menge der ausgewählten Testfälle. (●)
- Für jeden Testfall sind alle Eingabewerte (Bildschirmeingaben, Zustände des Systems) und Ausgabewerte (Bildschirmausgaben, Zustände des Systems) spezifiziert.

Ableiten von Funktionstests (IV) - Zweigtesten

Frage: Wie teile ich in Äquivalenzklassen ein?

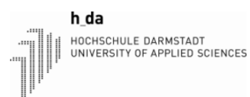
Welche Eingabedaten wählen Sie für die Testfälle?

- 3 Arten der Einteilung in Klassen:

  Testfälle so auswählen, dass folgendes gilt:
- **Aktivitätentest** (Aktivitätenüberdeckung)
  - ⇨ Jeder Kasten des Aktivitätsdiagramms wird 1 Mal durchlaufen
  - ⇨ So viel Durchläufe, dass jeder Kasten mind. 1 Mal durchlaufen wird

  Frage: wie oft muss ich den obigen Use-Case durchlaufen?
- **Zweigtesten**
  - ⇨ Jeder der Zweige des Aktivitätendiagramms wird mindestens ein Mal durchlaufen,
  - ⇨ d. h. jeder Pfeil im Aktivitätsdiagramm wird mindestens ein Mal durchlaufen.
  - ⇨ Jeder Durchlauf ist ein Testfall

  Frage: wie oft muss ich den obigen Use-Case durchlaufen?



Beispiel: Ableiten von Funktionstests (V): Aktivitatsdiagramm = Zweiggraph



Ableiten von Funktionstests (VI) - Zweigtesten

Eingabe	Soll-Zwischenergebnisse	Soll-Ergebnis
keine Eingabe für 30 Sek.		Abbruch
Abbruch-Taste gedrückt		Abbruch
richtige PIN eingegeben		Zugang frei
3x falsche PIN eingegeben	2 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch

Dies ist die Spezifikation der Äquivalenzklassen!

Frage: Ist dies ein Testplan?

Wie würde ein Testplan aussehen?

Testplan:

Vorzustand	Eingabe	Soll-Zwischenergebnisse	Soll-Ergebnis	Ist-Erg.
	keine Eingabe für 30 Sek.		Abbruch	
	Abbruch-Taste gedrückt		Abbruch	
Gültige PIN zu Kunden ist 4711	richtige PIN 4711 eingegeben		Zugang frei	
Gültige PIN zu Kunden ist 4711	PIN 1234, 2436, 7777 eingegeben	2 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch	

konkrete Werte!



Ableiten von Funktionstests (VII) - Pfadtesten

- ⇨ Jedes mögliche Szenario wird mindestens einmal getestet
- ⇨ d. h. jeder mögliche Pfad in den zu den Use-Cases gehörenden Aktivitätsdiagrammen wird mindestens ein Mal durchlaufen.
- ⇨ Anzahl der Wege wird schnell zu groß (⇨ Auswahl wichtiger Tests)

Frage: Was ist der Unterschied zwischen Zweig und Pfad?

Frage: Wie oft muss ich den obigen Use-Case durchlaufen?



Beispiel: Ableiten von Funktionstests (VIII): Pfadgraph

Aufforderung PIN-Eingabe
Kundenreaktion
keine Eingabe für 30 Sek.
Kunde bricht ab
PIN-Eingabe erfolgt
PIN-Validierung
PIN als richtig validiert
PIN falsch und < 3 Fehlversuche

Aufforderung PIN-Eingabe
Kundenreaktion
keine Eingabe für 30 Sek.
Kunde bricht ab
PIN-Eingabe erfolgt
PIN-Validierung
PIN als richtig validiert
PIN falsch und < 3 Fehlversuche

Aufforderung PIN-Eingabe
Kundenreaktion
keine Eingabe für 30 Sek.
Kunde bricht ab
PIN-Eingabe erfolgt
PIN-Validierung
PIN als richtig validiert
PIN falsch und >= 3 Fehlversuche

System gibt Zugang frei
Abbruch

Ein Pfadgraph stellt alle möglichen Pfade dar. Er hat keine Rückwärtssprünge.
Ein Szenario ist ein Weg vom Startknoten zu einem Endknoten.
=> Wenn jeder Pfeil mindestens ein Mal durchlaufen ist, sind alle Pfade durchlaufen, d. h. jedes mögliche Szenario ist mindestens ein Mal durchgespielt.

Frage: Wie viele Tests hätten wir, wenn wir PIN unendlich oft eingeben können und nur durch Abbruch raus kommen?



Ableiten von Funktionstests (IX) - Pfadtesten

- Spezifikation der Äquivalenzklassen:

Eingabe	Soll-Zwischenergebnisse / -ausgaben	Soll-Endergebnis
richtige PIN eingegeben		Zugang frei
ein Mal falsche PIN eingegeben, beim 2. Versuch richtige PIN	1 Mal Aufforderung der erneuten PIN-Eingabe	Zugang frei
zwei Mal falsche PIN eingegeben, beim 3. Versuch richtige PIN	2 Mal Aufforderung der erneuten PIN-Eingabe	Zugang frei
drei Mal falsche PIN eingegeben	2 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch
keine Eingabe für 30 Sek. statt 1. PIN-Eingabe		Abbruch
Abbruch statt 1. PIN-Eingabe		Abbruch
keine Eingabe für 30 Sek. statt 2. PIN-Eingabe	1 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch
Abbruch statt 2. PIN-Eingabe	1 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch
keine Eingabe für 30 Sek. statt 3. PIN-Eingabe	2 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch
Abbruch statt 3. PIN-Eingabe	2 Mal Aufforderung der erneuten PIN-Eingabe	Abbruch

Frage: Wie sieht der Testplan aus?

Ableiten von Funktionstests (X)

- Schwierigkeit bei den obigen Verfahren: Ich habe oft nicht das gesamte System als Use-Cases, sondern nur einige Scenarien definiert.

  => Ich muss mir überlegen, welche Scenarien ich zu Testzwecken noch zusätzlich spezifizieren sollte.

- Wie teste ich nun? Aktivitätentesten, Zweigtesten oder Pfadtesten?
  - es kommt darauf an, wie ausführlich man bei der Abnahme testen möchte.

    -> Testen aller Aktivitäten

    -> Testen einiger Zweige

    -> vollständiger Zweigtest

    -> Zweigtesten + einige Pfade aus dem Pfadgraphen oder

    -> vollkommener Pfadtest - falls überhaupt möglich

    (Wenn man z. B. die PIN-Eingabe beliebig oft wiederholen darf,

    => Pfadgraph ist unendlich groß => vollkommener Pfadtest nicht möglich)

    -> ungültige Äquivalenzklassen (Eingabe „A“) – Hier darf das System auch nicht abstürzen (Fehlertoleranz!)

    -> zusätzlich intuitive Testfälle (Eingabe 0 oder Blank),



Kontrollfragen zum Abnahmetest

- Welches Ziel haben Abnahmetests?
- Wann werden Abnahmetests festgelegt?
- Wie finden Sie Tests für funktionale Anforderungen?
- Kann es unendlich viele Testpfade geben?
- Was tun Sie, wenn es zu viele Testpfade gibt?
- Wie testen Sie nicht-funktionale Anforderungen?
- Welche Bedeutung hat der Abnahmetest im vertraglichen Sinn?
- Wie hängt der Abnahmetest mit den Anforderungen zusammen?

Können Sie jetzt ausgehend von einer Spezifikation Abnahmetests spezifizieren?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

5. Software-Architektur (Grob-Design)



Einordnung im V-Modell



Architektur einer Brücke



Quelle: R. Hahn

Diese "Brücke"

- kann von einer Person realisiert werden
- Einfacher Bauplan
- Einfache Arbeitsschritte
- Einfache Werkzeuge



Architektur einer großen Brücke



Quelle: Wikipedia, GFDL

"... every large software project is similar to the first attempt to build a flying-buttress construction cathedral."

(Richard P. Gabriel)

Die Funktion der großen Hänge-Brücke ist im Prinzip gleich, aber diese Brücke

- wird von vielen spezialisierten Personen(gruppen) realisiert
- in mehrjähriger Bauzeit
- komplizierte Baupläne
- aufwändige Arbeitsschritte
- viele spezielle Werkzeuge



Bau einer großen Brücke



- Es gibt mehrere professionelle Architekten
  - ⇨ möglichst mit Erfahrungen mit ähnlichen Bauvorhaben,
  - ⇨ die den Bau leiten und die Arbeiten koordinieren
- Es werden diverse Pläne erstellt,
  die die Brücke bzgl. verschiedener Aspekte zeigen
  - ⇨ Technische Zeichnungen zu Stahlkonstruktion
  - ⇨ Fundamentpläne
  - ⇨ Statikpläne mit Kräftevektoren
  - ⇨ Seilaufbaupläne
- Die Arbeiten müssen in Arbeitspakete zerlegt werden.
  - ⇨ Die Realisierung erfolgt später durch mehrere (spezialisierte) Teams

... und für
Software-Projekte???



Software-Projekt: Fahrerinformationssystem




Quelle: www.blaupunkt.com



- Umfang:
  - ⇨ Bereitstellung <u>neuer Funktionalität</u> (Spracheingabe, …)
  - ⇨ <u>Neuentwicklung großer Teile</u> (User Interface komplett)
  - ⇨ Auslieferung von <u>Zwischenständen</u>
- Team: ca. <u>50 SW Entwickler</u> + Management + Querschnitt
- Dauer: <u>2-3 Jahre</u>
- Budget: ca. <u>30 Mio. €</u>

Frage: Wie gehe ich solch ein Projekt an?

⇨ <u>Architektur für Software</u> ist notwendig!



Situation im Projekt

- ⇨ Es ist (mehr oder weniger) verstanden, was der Auftraggeber will
- ⇨ Es ist ein großes Projekt
- ⇨ Nun geht es an die Umsetzung in Software
- ⇨ aber wie gehe ich in einem großen Projekt vor?





Idee: Erstelle eine grobe Skizze des Systems!

- noch keine Details (Implementierung, Programmiersprache, Technik, …)
- Aufteilung in Teile (Pakete / Komponenten)
- Beschreiben der Verantwortlichkeiten der Pakete / Komponenten
- Beschreiben der Kommunikation der Pakete / Komponenten
- erste Evaluation der Funktionalität möglich

⇨ "Software-Architektur"



Definition

Die Software-Architektur eines Rechnersystems beschreibt die Struktur der Software des Systems, die aus Komponenten, deren extern sichtbaren Eigenschaften und Verantwortlichkeiten und deren Beziehungen besteht

(Len Bass et. al.)

Dilbert:

... Its a bunch of shapes connected by lines...

Eine von vielen Definitionen…
siehe auch:
http://www.sei.cmu.edu/architecture/definitions.html

Es kommt darauf an, die der Aufgabenstellung angemessenen Komponenten und Beziehungen zu finden!



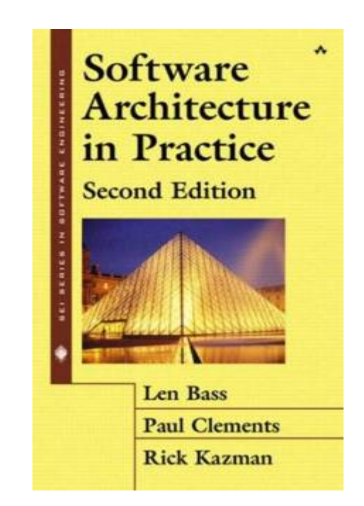
Lernziel Software-Architektur

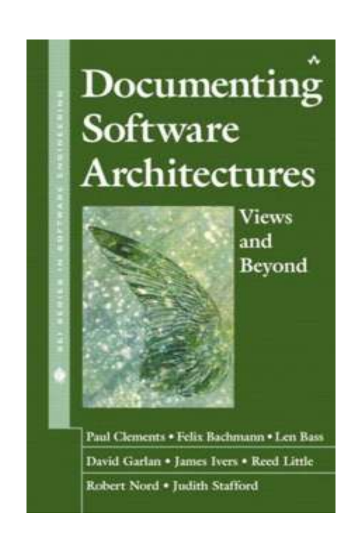
- Sie sollen in diesem Kapitel verstehen,
  - ⇨ warum eine Software-Architektur für ein großes Projekt wichtig ist
  - ⇨ was eine Software-Architektur ist
  - ⇨ dass es viele Architekturen zu einer Aufgabe gibt
  - ⇨ dass die verwendeten Techniken der SW-Architektur von der Anwendung abhängen
  - ⇨ dass die Qualitätsanforderungen entscheidend für die Architektur sind
  - ⇨ dass es bekannte Architekturstile mit bekannten Qualitätseigenschaften gibt
  - ⇨ dass es schwierig ist, eine gute Architektur zu entwickeln
  - ⇨ wie man eine Architektur darstellt

Anschließend verstehen Sie die Bedeutung einer SW-Architektur und ahnen, wie man eine SW-Architektur erstellt !

Literatur

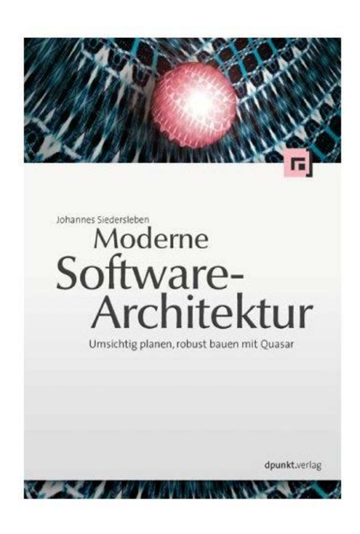
Len Bass et.al.:
Software Architecture in Practice;
Addison-Wesley; 2003





Paul Clements et.al.:
Documenting Software Architectures;
Addison-Wesley; 2002

Johannes Siedersleben:
Moderne Software-Architektur;
dpunkt; 2004

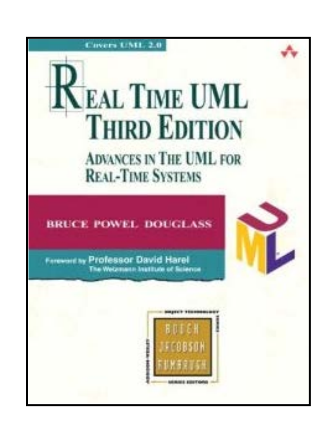




Literatur



Ralf Reussner, Wilhelm Hasselbring:
Handbuch der Software-Architektur,
dpunkt.verlag, 2006



[Douglass06]
Bruce Powel Douglass:
Real Time UML Third Edition,
Addison Wesley, 2006

[Kruchten95]
Philippe Kruchten:
Architectural Blueprints – The “4+1!
View Model of Software Architecture,
in IEEE Software 12 (6) November
1995, pp. 42-50



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

5.1 Architekturstile



Auf der Suche nach dem Architekturstil (=Architekturmuster) für ein Programm(-system) - Beispiel "KWIC"

- Aufgabe
  - ⇨ Wie sieht eine Architektur für folgendes (einfache) System aus?

The KWIC (KeyWord in Context) index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be ›circulary shifted‹ by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

"On the Criteria for Decomposing Systems into Modules", David Parnas. CACM, 1972

- Anwendung: z.B. Index im Unix Manual oder auch in Bibliotheken
  - ⇨ Suche nach Stichwörtern

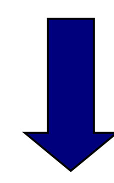
siehe auch: "Software Architecture: Perspectives on an Emerging Discipline"
Mary Shaw and David Garlan, Prentice Hall, 1996.



Funktionalität von "KWIC"

KeyWord in Context:
Jede Eingabezeile wird wortweise zyklisch geshifted und dann die Ergebniszeilen sortiert

- Eingabe
  - *Perspectives on Software Architecture*
  - *Computers in Context*



- Zwischenergebnis (zyklische Shifts)

Perspectives on Software Architecture
on Software Architecture Perspectives
Software Architecture Perspectives on
Architecture Perspectives on Software
Computers in Context
in Context Computers
Context Computers in

- Ausgabe (sortiert)

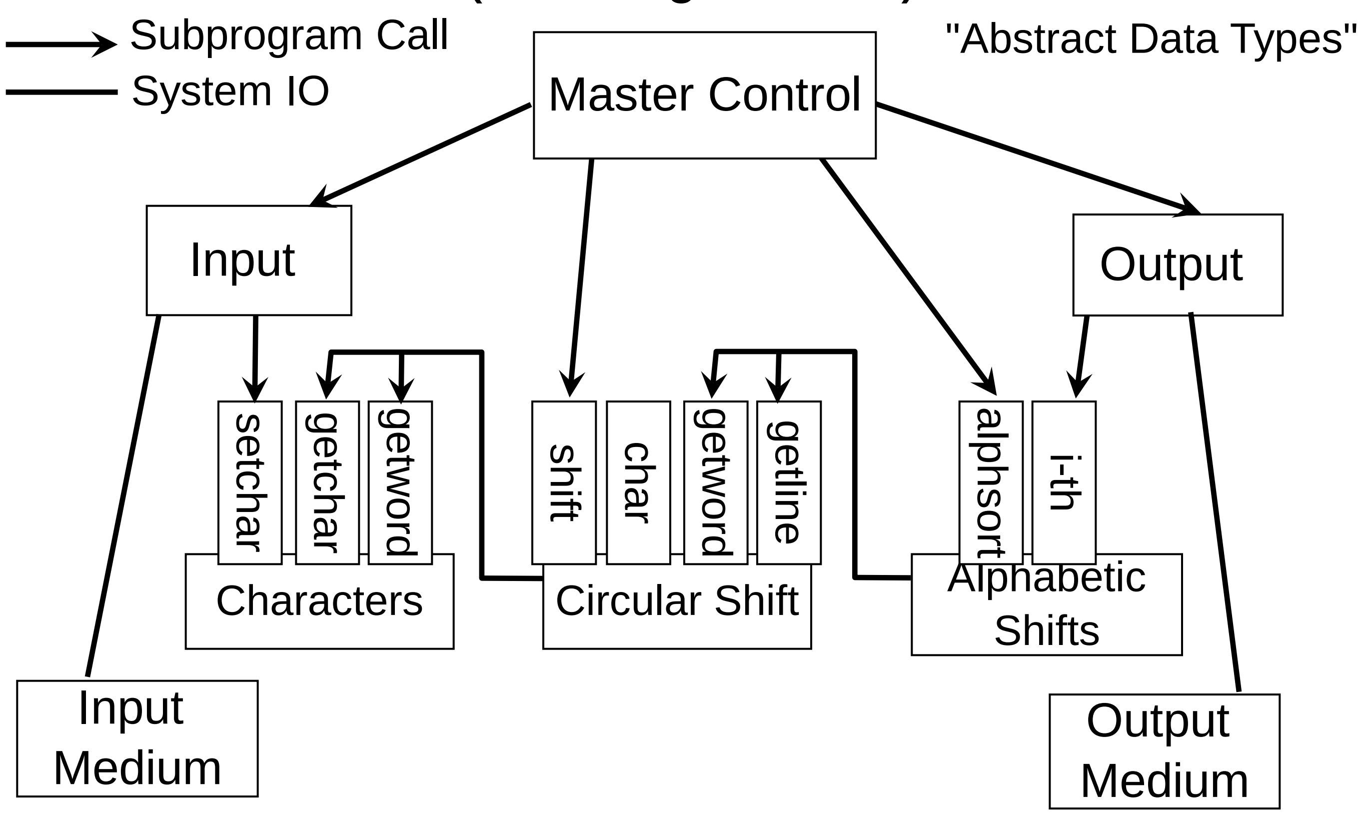
Architecture Perspectives on Software
Computers in Context
Context Computers in
in Context Computers
on Software Architecture Perspectives
Perspectives on Software Architecture
Software Architecture Perspectives on

KWIC: Prozeduraler Stil

- ⇨ Zerlegung in vier Basisfunktionen
- ⇨ Koordination durch ein Hauptprogramm, das sie sukzessive aufruft
- ⇨ Daten werden als gemeinsame Ressource von den Modulen verwendet
- ⇨ Keine Kapselung der Daten



KWIC: Objektorientierter Stil (zentral gesteuert)



- Kontrollfluss durch zentrale Koordination
- Datenkapselung
- Algorithmen und Datenrepräsentation der einzelnen Klassen können geändert werden, ohne die anderen zu beeinflussen



KWIC: Eventgesteuerter Stil

Implicit Invocation
Subprogram Call
System IO

IO Control

"Implicit Invocation"

Input | Circular Shift | Alphabetizer | Output

insert | delete | i-th
Lines

insert | delete | i-th
Lines

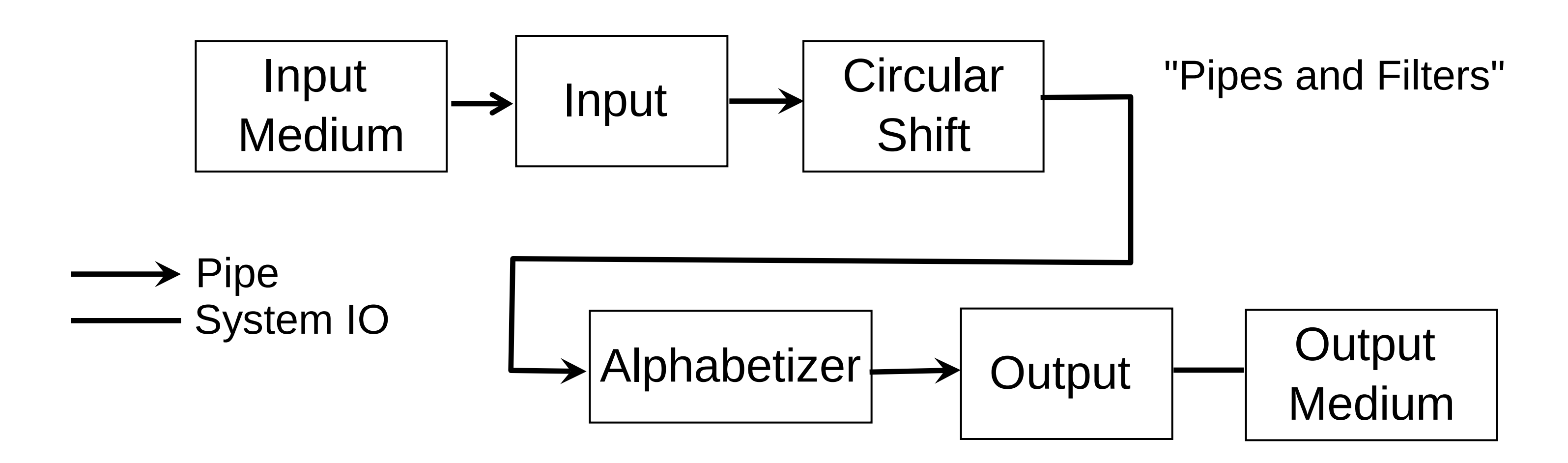
Input Medium

Output Medium

- ⇨ Keine zentrale Steuerung
- ⇨ Veränderung der Daten löst Events aus
- ⇨ Events führen zu Aktionen, die ihrerseits Events erzeugen
- ⇨ Modell „aktiver Daten“



KWIC: Pipes & Filters als Stil



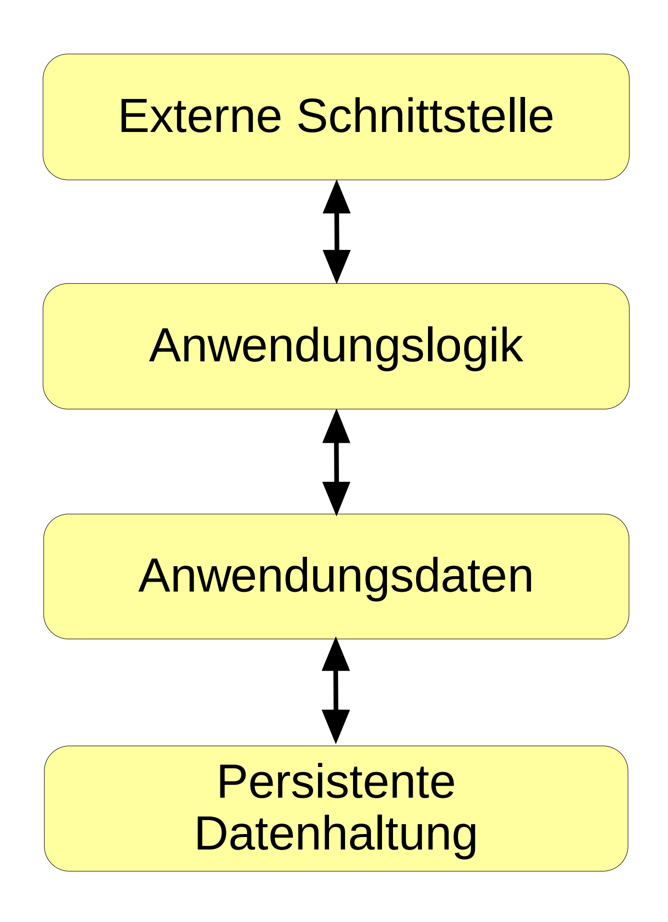
- Verteilte Steuerung: jeder Filter arbeitet, sobald Daten an seiner Eingabe anliegen
- Die Daten existieren nur temporär als I/O zwischen den Filtern
- Die Filter sind voneinander unabhängig, sie kommunizieren nur über die Pipes zwischen ihnen
- Datenfluss und Kontrollfluss fallen zusammen

Frage: Ist Compiler durch Pipes & Filters realisiert?

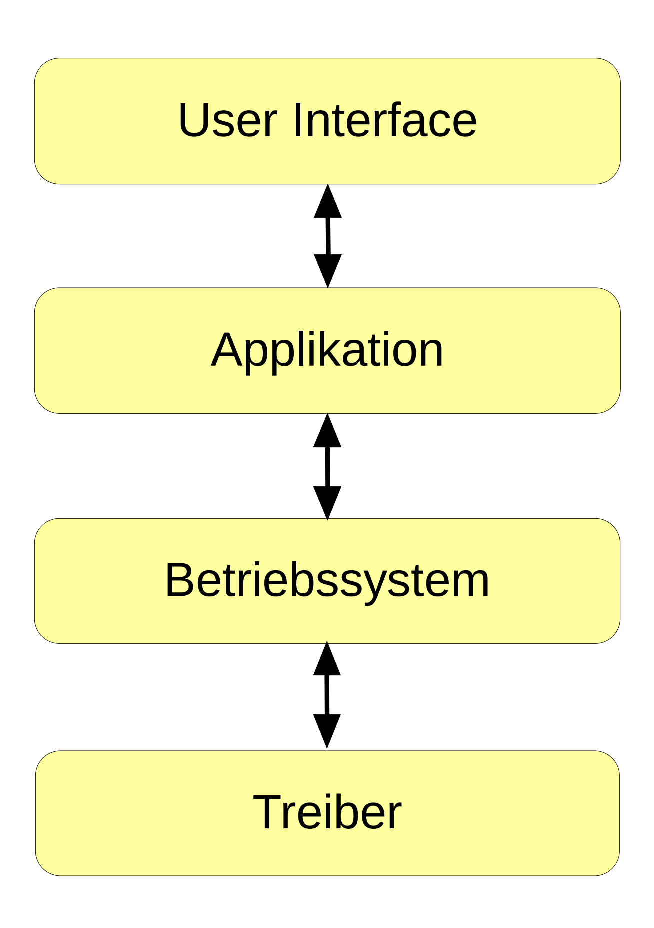


Architekturstil Pipes and Filters: Beispiel Compiler

Architekturstil 4-Schichten (Layers): Typischer Einsatz



Mit Hilfe des Layers-Stils lassen sich Anwendungen strukturieren, die in Gruppen von Teilaufgaben zerlegbar sind und in denen diese Gruppen auf verschiedenen Abstraktionsebenen liegen



Zuordnung der 4-Schichtenarchitektur zu Klassen-Stereotypen

Schicht	Klasse	**Stereotypname**	**Symbol**
Externe Schnittstelle	Schnittstellenklasse	<<boundary>>	
Anwendungslogik	Kontrollklasse	<<control>>	
Anwendungsdaten	Entitätsklassen	<<entity>>	
Persistente Datenhaltung			



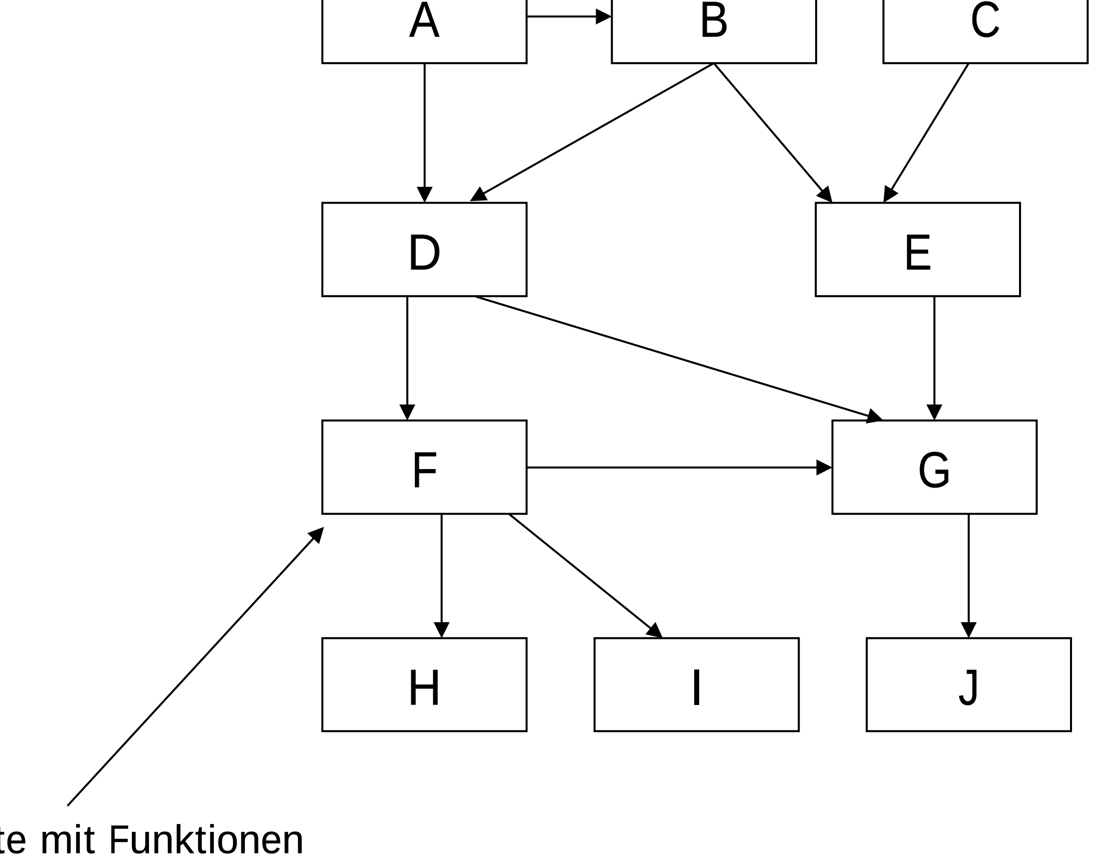
Beispiel: Aufruf von Funktionen in einer 4-Schichten-Software-Architektur

Schicht 1: Externen Schnittstelle

Schicht 2: Anwendungslogik

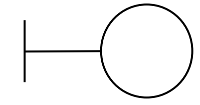
Schicht 3: Anwendungsdaten

Schicht 4: Persistente Datenobjekte





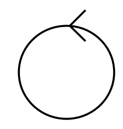
Was sind Schnittstellenklassen?



- Schnittstellenklassen sind zuständig für die Interaktion des System mit seiner Umgebung wie
  - ⇨ Benutzer
  - ⇨ anderen Systemen, die das entwickelte System benutzen
- Die Interaktionen beinhalten in der Regel
  - ⇨ die Auswahl und Aktivierung bestimmter (Anwendungs-)Funktionen sowie
  - ⇨ die Ein- und Ausgabe von Daten
- Weitere Aufgaben:
  - ⇨ Umsetzung der Interaktionen der Akteure in anwedungsinterne Ereignisse
  - ⇨ Prüfung von Eingabedaten
  - ⇨ Umsetzung von Eingabedaten in eine für Akteure verwertbare Form
  - ⇨ Ausgabe von Daten
- Wir können aus Use-Cases Schnittstellenklassen ableiten
  - ⇨ diese sind zuständig für die in dem Use-Case spezifizierte Kommunikation mit dem Akteuren.
  - ⇨ Klassenname: Use-Case-Name mit angehängten AF [Wi05] oder UC:
  - ⇨ Oft haben Use-Cases auch Kontrollfunktion. In [Wi05]:
- Schnittstellenklassen sind oft versteckt in GUI-Frameworks

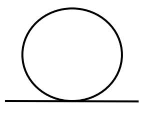
GeldAbhebenUC

Was sind Kontrollklassen?



- Kontrollklassen beinhalten die anwendungsfallbezogene (meistens mehrere Instanzen von Entitätsklassen betreffende) fachliche Logik.
- Sie steuern die Abläufe
- Sind Vermittler zwischen Schnittstellen- und Entitätsklassen und kontrollieren, ob die von der Schnittstellenklasse angenommenen Eingaben und Modifikationen der Akteure auf den Entitätsklassen ausgeführt werden dürfen
- (Falls Anwendungslogik trivial: Auch direkte Kommunikation zwischen Schnittstellen- und Entitätsklassen. – In Schnittstellenklassen ist dann evtl. anwendungsfallbezogene fachliche Logik integriert)

Was sind Entitätsklassen?



- Enthalten Daten.
- Enthalten die fachliche Logik, die die einzelnen Instanzen der Entitätsklasse betreffen. (Die fachliche Logik, die nur eine Instanz einer Entitätsklasse betrifft, sollte nicht in Kontrollklassen untergebracht sein – vor allem nicht in einer großen Kontrollklasse, die die gesamte Anwendungslogik enthält! = Blob)
- Jede Entitätsklasse ist dafür verantwortlich, dass ihre Instanzen nur die gemäß Geschäftsregeln erlaubten Zustände annehmen (Setter, Getter).
- u. a. werden oft die in der Phase Anforderungsermittlung ermittelten Domain- / Analyseklassen in Entitätsklassen überführt.

Persistente Datenhaltung

- Diese Schicht enthält die persistent gespeicherten Daten.
- Entweder
  - ⇨ werden Instanzen von als persistent gekennzeichnete Klassen gespeichert (OODBMS)

    oder
  - ⇨ die Daten werden in z. B. relationalen DBMS abgelegt.

    (Mit Umsetzungsschicht, die für die Transformation der Klassensicht in relationale Sicht und umgekehrt zuständig ist. Z. B. Hibernate)



Welchen Architekturstil wende ich für meine Aufgabe an?

- Der "richtige" Architekturstil
  - ⇨ ist abhängig von der Anwendung / Teilanwendung / Hardware
    z. B. bei KWIC: Passen Daten in Hauptspeicher
  - ⇨ es können auch mehrere Stile kombiniert werden
    z. B. ereignisgesteuert, objektorientiert und 4-Schichten
  - ⇨ ist abhängig von den Anforderungen
    - Änderungsfreundlichkeit bezüglich Datenstruktur, Funktionalität,
    - Performance (Platz, Laufzeit), ...

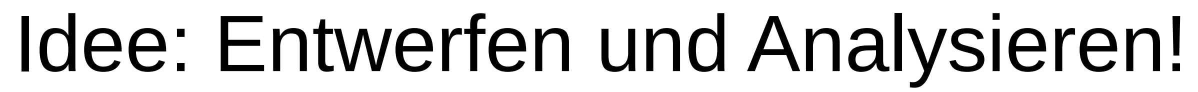
- Jeder Architekturstil hat Vorteile und Nachteile
  - ⇨ die resultierende Implementierung ist stark verschieden

⇨ Die bekannten Eigenschaften eines Architekturstils helfen bei der Auswahl !



Vorgehen beim Grobentwurf (Finden der richtigen Architektur)







- ⇨ Durch Entwerfen von Komponenten, Beziehungen und Verantwortlichkeiten (z.B. OOAD: Komponentendiagramme)
- ⇨ iteratives Analysieren der Anforderungen mit Szenarien ("Welche Änderungen werden erwartet? Sind wahrscheinlich?) vgl. OOAD: Sequenzdiagramme, Kommunikationsdiagramme auf Klasseninstanz-Ebene
- ⇨ Erleichterung durch (Wieder-)Verwendung von bewährten "Mustern" mit bekannten Eigenschaften ("Architektur-Stile")

⇨ Erfahrung hilft dabei sehr !



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

5.2 Darstellung der SW-Architektur



Darstellung der Architektur

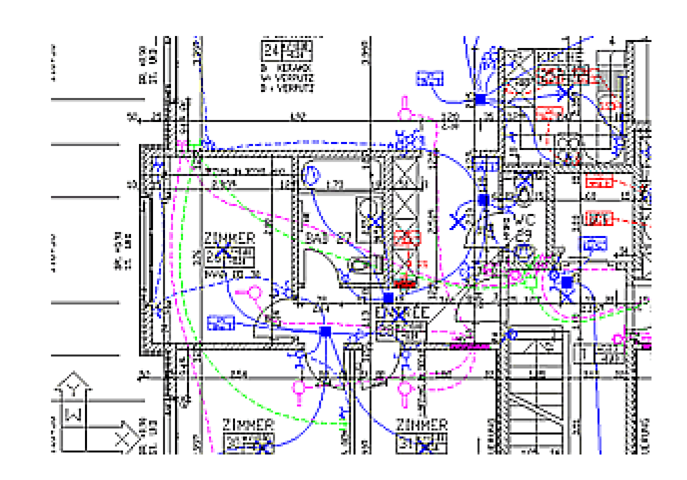
Was macht ein Architekt? Viele Pläne!



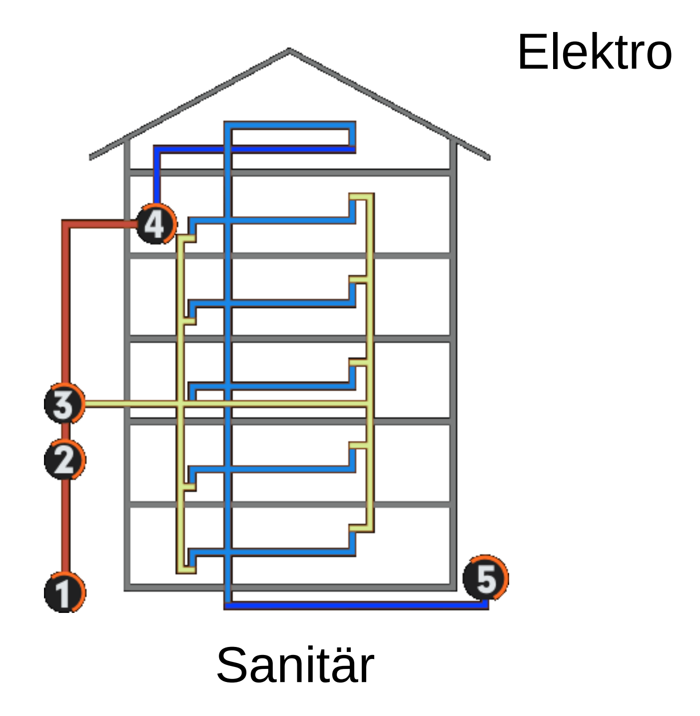
Endkunde



Bauarbeiter



Elektro



Sanitär



Sichten der Software-Architektur

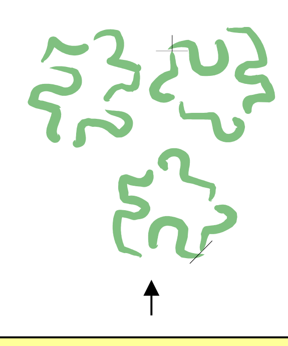
- Beim Hausbau gibt es viele verschieden Pläne ("Sichten")
  - ⇨ Raumaufteilung (Grund- und Seitenriss)
  - ⇨ Elektroinstallation und Datennetz
  - ⇨ Sanitäre Installation
  - ⇨ …
- Bei der Software gibt es ebenfalls verschiedene Sichten
  - ⇨ Logische Zerlegung
  - ⇨ Verteilung auf Zielsysteme
  - ⇨ Quelldateien
  - ⇨ Prozesse / Threads
  - ⇨ …

⇨ komplexe Systeme werden in mehreren Sichten ("Views") dargestellt
⇨ Im konkreten Fall wählt man die "passenden" Sichten
⇨ Das Wesentliche der Architektur lässt sich in der Regel nicht in einer einzigen Sicht darstellen
⇨ Es gibt von unterschiedlichen Autoren viele verschiedene Einteilungen in Views

Logische and Physikalische Architektur

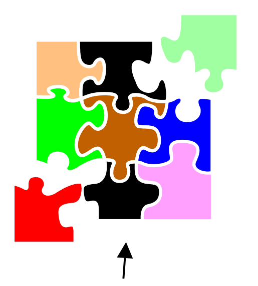
nach [Douglasss04]: Real Time UML, Addison Wesley, 2004

Logical Architecture:



Organisiert Dokumente zum System (Jedes Dokument taucht nur 1 Mal auf)

Physical Architecture:



Stellt das Zusammenspiel der Systemteile zur Laufzeit dar, wie Subsysteme, Komponenten und Tasks

Folgende Namen hat diese Sicht in der Literatur:

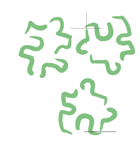
- ***Paket-Sicht*** */ Domänen Sicht / Logische Architektur*

Folgende Namen haben diese Sichten in der Literatur:

- ***Subsystem- und Komponenten-Sicht*** */ Logische Sicht*
- ***Prozess-Sicht*** */ Thread-Level / Concurrency and Resource View*
- ***Einsatz-/*** *Deployment-/ Physische* ***Sicht*** *(HW-Komponenten)*
- ***Implementations****- / Dateien-****Sicht***
- ***Anwendungs-Sicht*** *(nach [Kruchten95])*
- ....



Paket-Sicht / Domänen Sicht / Logische Architektur



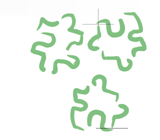
- Organisiert Dinge die zur Design Zeit existieren.
- System wird in Pakete (Domänen) gegliedert.
- Ein Paket ist ein Bereich mit Dingen (Ergebnisse von Arbeitsprozessen = Artifakte), normalerweise mit einem eigenen Vokabular (Namensraum bei Programmen)
- Jeder Artifakt ist genau einem Paket zugewiesen
- Pakete werden oft in Abhängigkeits-Hierarchien angeordnet.

  Gliederung des Gesamtpakets zum System (Menge aller Artifakte) in Schichten,

  z. B. Paket für Anforderungsanalyse, Design, Implementierung (siehe auch linkes Fenster im Case-Tool des Praktikums).

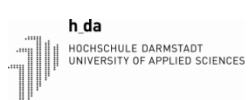
  - ⇨ Unter Anforderungsanalyse: Für jedes Use-Case-Diagramm ein (Unter-)Paket
  - ⇨ Unter Design: Aufteilen der Klassen des Klassendiagramms auf versch. Pakete. Eine Generalisierungshierarchie befindet sich normalerweise in einem Paket
  - ⇨ Unter Implementierung: mehreren Pakete (getrennte Name-spaces) für den Code.





Paket-Sicht (gemäß UML, siehe Vorl. OOAD W.Weber) / Domänen Sicht / Logische Architektur

- Hierarchie von Paketen (dargestellt durch Schachtelung)
- import: Das Quellpaket (Paket, bei dem Pfeil beginnt) kann auf alle public-Elemente des Zielpaketes ohne Qualifizierung (Zielpaket::) zugreifen.
  -> Programmier-Vorl.: Paket std



Paket-Sicht / Domänen Sicht / Logische Architektur (in Anlehnung an [Douglass06])

- Beispiel mit Darstellung der Klassen und deren Beziehungen in den Paketen

Hold classes and types for Alarm Management

Alarm Domain

Critical Alarm

Alarm Filter Policy

Informational Alarm

Alarm

Alarm Manager

<<import>>

User Interface Domain

Display Control

Text

Window

Text List

<<import>>

Hold classes and types for User Interface

Hold classes and types that represent hardware

Hardware Domain

Speaker

Button

Pushbutton Turnknob

Knob



Physikalische Architektur



- Physikalische Architektur stellt Zusammenspiel der Systemteile dar, wie Subsysteme, SW-/HW-Komponenten und Tasks
- Einige Subsysteme verwenden nur Artifakte aus einer Domäne (Paket), andere benutzen Teile aus verschiedenen Domänen
- Die Physikalische Architektur beschreibt auch die Nutzung der Infrastruktur des Systems
- Folgende Sichten sind in der Literatur beschrieben:
  - Subsystem- und Komponenten-Sicht / Logische Sicht
  - Prozess-Sicht / Thread-Level / Concurrency and Resource View
  - Einsatz-/ Deployment-/ Physische Sicht (was läuft auf welcher HW-Komponente)
  - Implementation- / Dateien-Sicht
  - Anwendungs-Sicht (nach [Kruchten95])
  - ...

-> Tafelbild





Ebenen der Abstraktion: Subsysteme, Komponenten, Threads (siehe [Douglass06])

Actor
Actor

System Level

<<subsystem>>
<<subsystem>>
<<subsystem>>

Systems Engineering Level

Electronic
Software
Mechanic

Multidisciplinary Level

Component
Component
Component
Component

Software Component Level

<<active>>
<<active>>
<<active>>

Thread Level



Subsystem-Sicht, Komponenten-Sicht / Logische Sicht



- Die Subsystem- und Komponenten-Sicht identifiziert die großen Teile des Systems und wie sie zusammenpassen (Darstellung z. B. durch das Komponenten-diagramm der UML)
- Es gibt keine klare Unterscheidung zwischen Subsystemen und Komponenten
- Wenn wir annehmen, dass Subsysteme die größten Teile eines Systems sind, dann sind Komponenten Teile von Subsystemen oder Komponenten
- Ein Subsystem / eine Komponente ist ein sehr großes Teil des Systems, das über Komposition kleinere Teile des Systems enthält, die die Arbeit des Subsystems / der Komponenete erledigen
- Das Kriterium für die Aufnahme in ein Subsystem / Komponente ist ein gemeinsamer zu erfüllender Zweck (starke Kohäsion)
- Ein Subsystem / eine Komponente stellt gut definierte Interfaces zur Verfügung und delegiert die Dienste zu internen versteckten Teilen.
- Die konkrete Kommunikation zwischen instanziierten Komponenten beschreibt den Ablauf des Systems (Darstellung z. B. durch Kommunikations- oder Sequenz-diagramme der UML)





Subsystem-Sicht (in Anlehnung an [Douglass06]) /Logische Sicht

Darstellung der Subsysteme und der Beziehungen zwischen den Subsystemen.
(Könnte auch mit UML-Komponentendiagramm dargestellt werden)

Air Traffic Control System

Primary Radar

Secondary Radar

National Airspace System

Radar_Subsystem <<subsystem>>

Antenna Subsystem <<subsystem>>

NAS_Interface <<subsystem>>

AircraftSpace Subsystem <<subsystem>>

Display_Subsystem <<subsystem>>

Alarm_Subsystem <<subsystem>>

AircraftCommunication Subsystem <<subsystem>>

Scheduling Subsystem <<subsystem>>

Controller

Aircraft

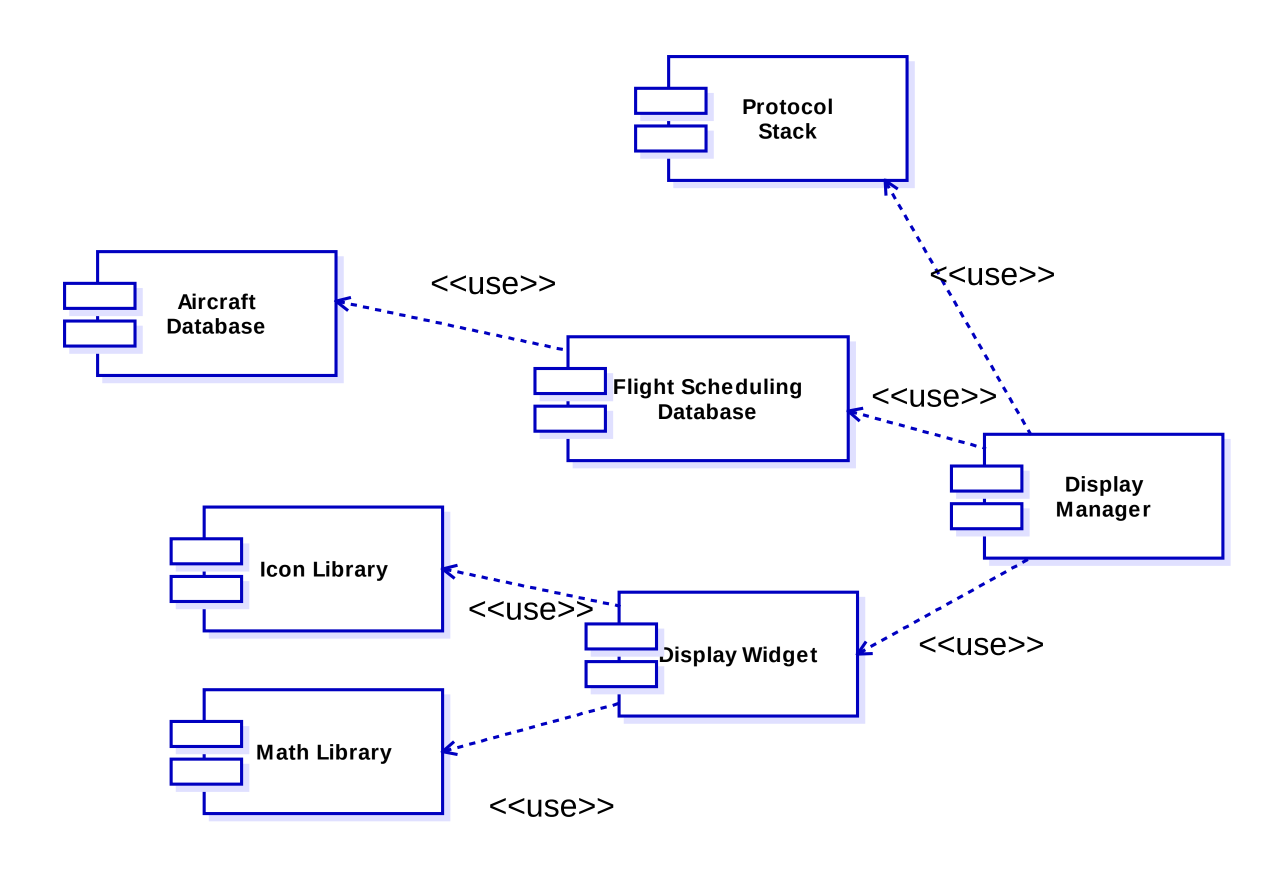
Flight Planning System



Komponenten-Sicht (in Anlehnung an [Douglass06]) /Logische Sicht



Komponenten des Display-Subsystems





Komponenten- / Logische Sicht (in Anlehnung an [Kruchten95])



cod Autoradio

«component»
Audio Manager

Audio Requests

1

1..*

«component»
Audio Source Control
{abstract}

«component»
Traffic Announcement

«component»
Radio

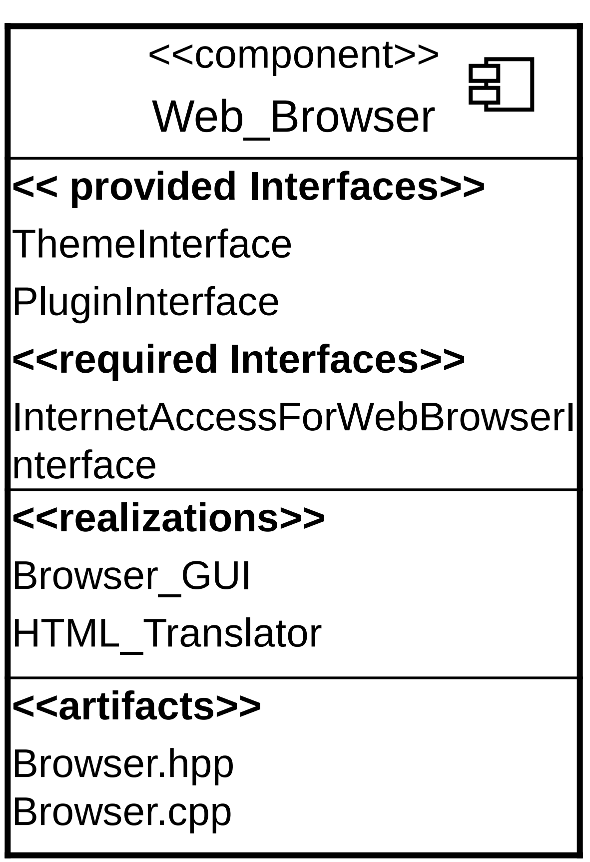
«component»
Cassette

«component»
CD



Komponenten-Sicht: Komponentendiagramm (gemäß UML) - Beispiel





White-Box
(mit Interna)

InternetAccessForWebBrowserInterface

<<component>> Internet_Access

ThemeInterface

<<component>> Web_Browser

<<component>> Theme

<<component>> Plugin

Black-Box
(Überblick)

PluginInterface

Plugin greift auf Webbrowser zu

<<component>> Web_Browser

<<Interface>> PluginInterface

<<use>>

<<component>> Plugin





Komponenten-Sicht: Komponentendiagramm (gemäß UML) - Darstellungselemente

[UML2 glasklar]

Komplexer Port
Implementierte Schnittstelle
Komponente
«component»
Komponente1
«component»
Komponente2
Bestandteil3
1..*
Bestandteil4
Klasse
Bestandteile einer Komponente
Kompositions-konnektor
Allgemeine Abhängigkeitsbeziehung
Realisierungs-beziehung
«artifact»
Artefakt1
Bestandteil1
Bestandteil2
«specification»
specDoc
Komponente
Verwendungs-beziehung
«use»
«subsystem»
Komponente3
«component»
Komponente4
«artifact»
Komponente6
«manifest»
Port
Implementierungs-beziehung
Delegationskonnektor
Benötigte Schnittstelle
Delegationskonnektor






Komponenten-Sicht: Komponentendiagramm (gemäß UML) - Darstellungselemente

- Port: Kommunikationspunkt (interaction point). Z. B. zur Gruppierung mehrerer Schnittstellen (die evtl. nur gewissen anderen Komponenten zur Verfügung gestellt werden).
- In einer Komponenete können auch die in ihr enthaltenen Klassen und deren Beziehungen untereinander dargestellt sein (auch möglich in einem Kompositionsstrukturdiagramm (S. 191f in [UML2glasklar]) oder Structured Class ([Douglass06])).





Komponenten-Sicht: Komponentendiagramm (gemäß UML) - Darstellungselemente

[UML2 glasklar]

Paginierer
Satzprogramm
«component»
Teilnehmerverwaltung
Sortierter Zugriff
Wahlfreier Zugriff
«component»
Listengenerator
Speichermedium
«manifest»
«artifact»
Lister_v2.0.ear
«component»
Datenbankmanagementsystem
«subsystem»
Ausgabesystem
«component»
XML-Schnittstelle
«manifest»
«artifact»
Oracle
Hardwarezugriff
Hardwarezugriff
Dateizugriff
«component»
Hardwarezugriffsschicht

Komponenten-Sicht: Komponentendiagramm (gemäß UML) - Beispiel – Erklärung der Elemente



- Teilnehmerverwaltung bietet einen Port mit den Schnittstellen „Sortierter Zugriff“ und „Wahlfreier Zugriff“ an und benötigt die Schnittstelle Speichermedium.
- Speichermedium wird bereit gestellt durch Komponente Datenbankmanagementsystem.
- Datenbankmanagementsystem besteht aus 2 Komponeneten
- Listengenerator benötigt die Schnittstelle „Sortierter Zugriff“
- Paginierer und Satzprogramm sind Unterkomponenten von Listengenerator, die von Listengenerator benötigt werden
- Die Komponenete Listengenerator wird durch Lister_v2.0.ear realisiert.

Frage: Was könnten bei Cocktail-Pro Komponenten sein?



Prozess-Sicht / Thread-Level / Concurrency und Resource View



- Richtet sich auf das Management von Resourcen und die Nebenläufigkeitsaspekte der Systemausführung
- Die Sicht beschreibt die Synchronisation von Threads und die gemeinsame Nutzung von Resourcen
- Dies ist insbesondere für Embedded Systeme und Systeme zur Prozesssteuerung eine der wichtigsten Sichten





Prozess-Sicht / Thread-Level (in Anlehnung an [Douglass06]) / Concurrency and Resource View

Darstellung von Beziehungen zwischen Threads

Built in Test Thread
<<Task>>

Separation Distance Monitor Thread
<<Task>>

Alarm Filtering Thread
<<Task>>

Aircraft Flight Plan Monitor Thread
<<Task>>

CmdRequest Queue
<<Message Queue>>

Alarm Annunciation Thread
<<Task>>

Speaker

Lock
<<Semaphore>>

Alarm List
<<Resource>>

ListView
<<Resource>>

Alarm

TextView

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 * *

Concurrency view in the Alarm subsystem



Prozess-Sicht (in Anlehnung an [Kruchten95]) / Thread-Level / Concurrency and Resource View

Darstellung der Kommunikation zwischen Threads (Prozessen)

cd PHP Request

<<process>>
Mozilla : HTMLClient

1: HTML Get (URI) →

4: return(HTMLPage) ←

<<process>>
Apache : Webserver

2: PHP Parse(HTML_Code) ↓

3 return (HTMLPage) ↑

<<process>>
tempPHP : PHP



Einsatz- / Deployment- / Physische Sicht

- Betrachtet, wie die Softwarearchitektur auf die <u>physikalischen Komponeten</u> abgebildet wird (<u>Rechnerknoten im Netz</u>, <u>Prozessoren</u> etc.)
- Viele <u>Embedded Systeme</u> bestehen aus <u>Multiprozssorsystemen</u> mit vielen <u>unterschiedlichen Hardwareeinheiten</u>. (Manche Autos haben mehr als 50 Prozessorknoten und mehrere hundert Sensoren)



Einsatz- / Deployment-Sicht (in Anlehnung an [Douglass06]) / Physische Sicht

Verteilungs-/Deployment-diagramm:

Network

Main Data Aqu Processor

Backup Data Aqu Processor

Flight Recorder

Scheduling

Backup Display Controller

Display Controller

Radar Bus

Tape System

Mirrored Tape System

NAS Interface Device

Terminal

Primary Radar

Secondary Radar

GPS Sensor



Einsatz- / Deployment- / Physische Sicht (in Anlehnung an [Kruchten95])

csd Audio-Video-Hardware

Audio-Video-System

Audio/Videosystem (Vorne)
Unterstützt TV, DVD, MP3, CD-Audio

CD1 Control Send
CD Control
CD1 In
CD Out
CD2 Control Send
CD2 In

CD/DVD Player (vorne)
Liest DVD, MP3, CD-Audio

CDC Control
CD Out

CD Changer
Liest CD-Audio

AV Out
AV In
TV In (FBAS)

Videoterminal
Monitor – zeigt eine Videoquelle an

TV Out (FBAS)

TV Tuner
Empfängt TV Signal und liefert ein TV Video

Antenne



Implementations-Sicht (in Anlehnung an [Kruchten95])



- Komponentendiagramm mit Dateien, Repositories und deren Beziehungen untereinander

cod CocktailPro2006

«artifact» CocktailPro.cpp

«use»

«component» Rezepturprozessor

«artifact» Rezepturprozessor.hpp

«use»

«artifact» Rezepturprozessor.cpp

«use»

«artifact» Rezeptbuch.hpp

«artifact» Rezeptbuch.o

«use»

«component» Dosierer

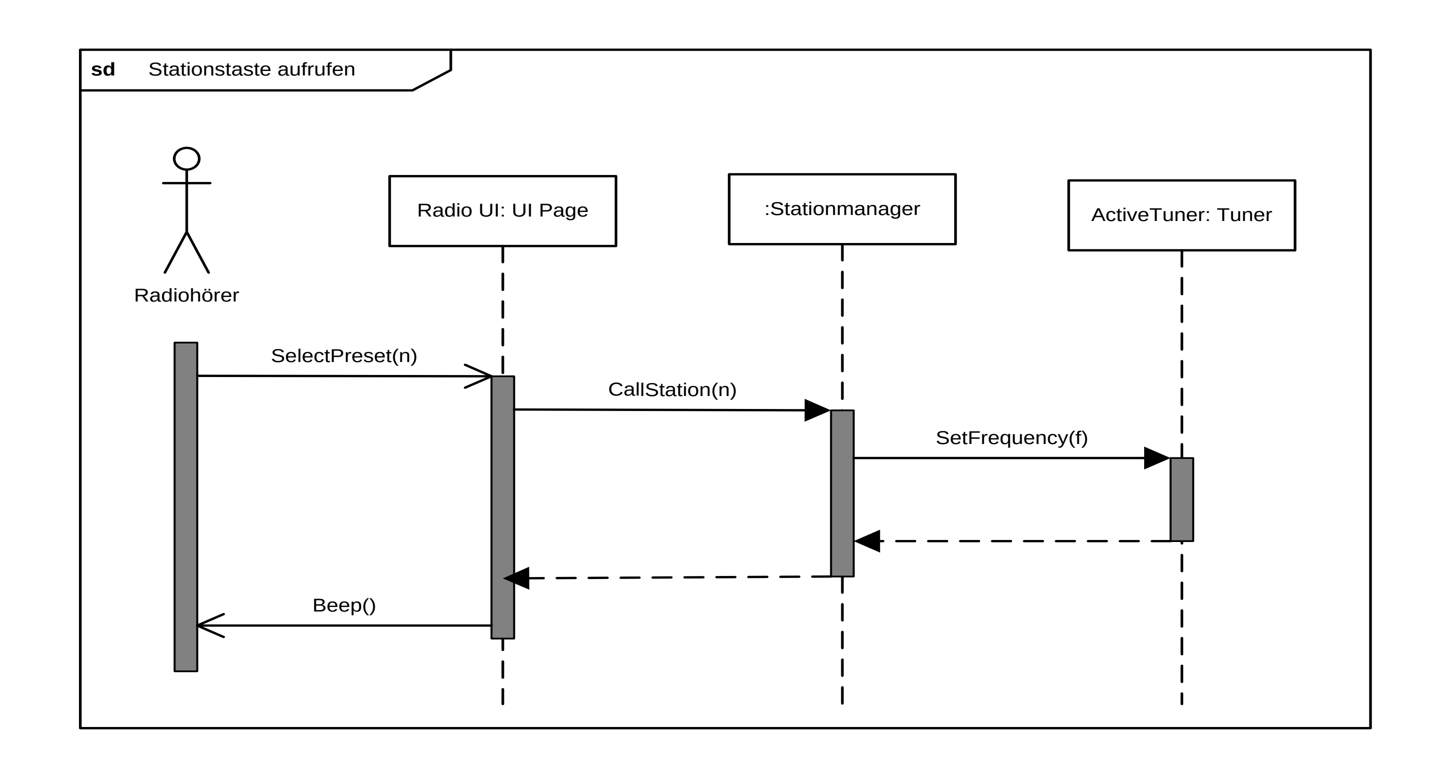
«artifact» Dosierer.hpp

«artifact» Dosierer.cpp



Anwendungs-Sicht (nach [Kruchten95])

- Darstellung des Verhalten des Systems, der Interaktion z. B. zwischen Komponenten-Objekten
- Umsetzung: Sequenz-, Kommunikationsdiagramme u. ä.



Dokumentation und Tools

- Die UML bietet diverse Möglichkeiten um Architektursichten darzustellen. Zum Beispiel:
- Paket-Sicht - Logische Architektur – Domänen-Sicht :Paketdiagramm
- Subsystem- und Komponenten- / Logische Sicht:
  - ⇨ Statik: Komponentendiagramm, Klassendiagramm, Kompositionsstrukturdiagramm (Structured Classes),
  - ⇨ Dynamik: Kommunikationsdiagramm, Sequenzdiagramm
- Prozess-/ Thread-Sicht / Concurrency and Resource View:
  - ⇨ Zustandsdiagramm, Kommunikationsdiagramm, Sequenzdiagramm (Message Sequence Chart), Aktivitätsdiagramm, Timing Diagramm
- Einsatz-/ Deployment-/ Physische Sicht :
  - ⇨ Verteilungs-Diagramm, Komponentendiagramm
- Implementations-Sicht / Dateien-Sicht :
  - ⇨ Komponentendiagramm

Welche Diagramme verwendet werden (dürfen), legt der Architekturprozess fest!



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

5.3 Zusammenfassung SW-Architektur



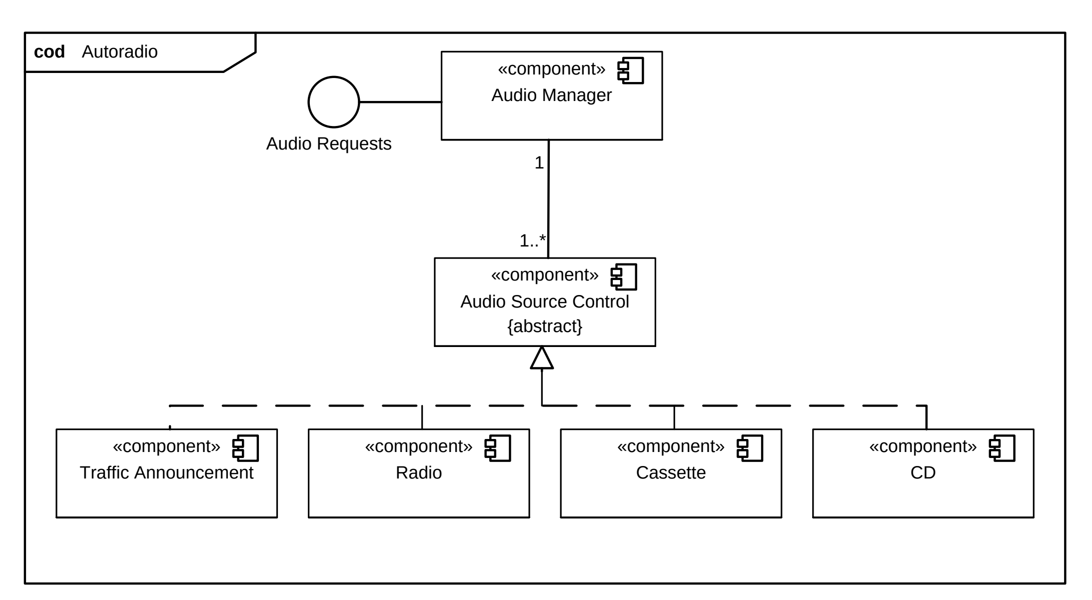
Beispiel zur Softwarearchitektur (I)

- Stellen Sie sich vor, Sie sollen ein einfaches Autoradio entwickeln
  - ⇨ mit Radio inklusive Verkehrsfunk, Kassette und CD
  - ⇨ mit den üblichen Verstärkerfunktionen
- Was sollten Sie nun in der Architektur festlegen?
  - ⇨ alles was festgelegt werden muss, bevor die Bestandteile des Systems einzeln bearbeitet werden können:
    - die Aufteilung des Systems in Komponenten (z.B. Radio, CD, usw.) mit Verantwortlichkeiten
    - die Schnittstellen zwischen diesen Komponenten untereinander und zur Außenwelt
    - die Verteilung der Software auf Dateien
- den Aufbau des Systems als Hardware
    - die Umsetzung von kritischen Vorgängen (z.B. Timingverhalten beim Überblenden)
    - die Verteilung der Software zur Laufzeit auf Hardware (Entwicklungsrechner, Target)
    - die Umsetzung von (komponentenübergreifenden) Use Cases zur Veranschaulichung (z.B. "Wechsel von CD auf Radio" betrifft CD, Amplifier und Radio)

Logische Sicht, Implementierungs-Sicht, Prozess-Sicht, Physische Sicht, Anwendungssicht!



Beispiel zur Softwarearchitektur (II)



- Ein Ergebnis der Architektur für das Autoradio könnte sein:
  - Diverse Komponentendiagramme, welche die Aufteilung des Systems in Komponenten und Schnittstellen zeigen (z.B. Audio Source und Audio Manager) und die Verantwortlichkeiten dokumentieren
  - Verteilungsdiagramme, welche die Hardware und die jeweils darauf laufende Software aufzeigen (Soundprozessor, Entwicklungsrechner, Target etc.)
  - Timing-Diagramme, welche die Umsetzung von kritischen Vorgängen mit Prozessen und Threads zeigen z. B. für Zeitverhalten beim Überblenden
  - Sequenzdiagramme, welche die Umsetzung von Use Cases skizzieren, die mehrere Komponenten betreffen (z.B. Audioquelle wechseln)
  - weitere Dokumente und Diagramme je nach Entwicklungsprozess

Die Anzahl und Art der Diagramme hängt stark vom Entwicklungsprozess, dem Projekt, dem Team und dem Kommunikationsbedarf ab!



Nutzen einer SW-Architektur

- für den Entwicklungsprozess
  - ⇨ trifft weitreichende, frühe Designentscheidungen
  - ⇨ weist Machbarkeit von Softwarefunktionalität nach
  - ⇨ macht Vorgaben für Design und Implementierung
  - ⇨ strukturiert den Entwicklungsprozess
  - ⇨ bietet Kommunikationsplattform
- für die Planbarkeit
  - ⇨ liefert die Einheiten der Planung für Design, Implementierung, Test und Wartung
- für die Wettbewerbsfähigkeit
  - ⇨ reduziert Kosten durch Wiederverwendung im Großen
  - ⇨ erhöht Flexibilität durch Betrachtung von Varianten ohne Umsetzung

Ohne eine SW-Architektur ist die Komplexität eines großen Projekts nicht zu bewältigen!



Auswirkungen von Fehlern in der SW-Architektur

- Falsche Requirements:
  - ⇨ Unterschätzung der Bedeutung / Übersehen eines wichtigen Requirements
    - Schwerwiegende Änderungen in späteren Phasen / keine Wiederverwendung
  - ⇨ Überschätzung der Bedeutung
    - Unnötige Arbeit und Kosten
  - ⇨ Falsche Gewichtung der Requirements gegeneinander
    - Falsche Ausrichtung der Architektur / Verfehlung des Projektziels
- Fehler in der Struktur aus Komponenten und deren Beziehungen:
  - ⇨ Ungeschickte Aufteilung bzw. Festlegung der Verantwortlichkeiten
    - Kleine Änderungen haben große Auswirkung
  - ⇨ Zu grobe Struktur
    - Keine ausreichenden Vorgaben für die Entwicklung / Übersehen von Problemen
  - ⇨ Zu feine Struktur
    - Ineffiziente Architektur-Arbeit

Architekturentwicklung ist immer eine Gratwanderung!



Kontrollfragen zur Software-Architektur

- Warum ist eine Software-Architektur für ein großes Projekt wichtig?
- Was passiert, wenn die Bedeutung einer Anforderung unterschätzt wird?
- Welche Aspekte des Entwurfs drückt man in der SW-Architektur aus?
- Wie wirkt sich die Verwendung eines Architektur-Stils aus?
- Hängt die SW-Architektur von der Programmiersprache / dem Betriebssystem ab?
- Wie dokumentieren Sie eine SW-Architektur?

Ahnen Sie jetzt wie man eine SW-Architektur erstellt?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

6. (Fein-)Design



Einordnung im V-Modell



Lernziel Design

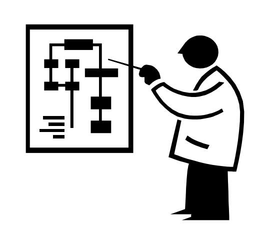
Frage: Was wissen Sie schon über das Design von Software?
Welche Modelle kennen Sie bereits zum Design von SW (aus OOAD)?
Welche Grundprinzipien, Welche Regeln des Designs kennen Sie?

- Sie sollen in diesen Kapitel,
  - ⇨ verstehen was im Design passiert
  - ⇨ weitere Techniken kennen lernen, die einen guten Entwurf fördern
  - ⇨ wiederholen, welche Grundprinzipien beim Entwurf gelten
  - ⇨ wiederholen, welche Regeln es zum Entwurf von Klassen und Assoziationen gibt
  - ⇨ verstehen warum man Design-Patterns verwenden sollte
  - ⇨ verstehen was ein Anti-Pattern ist
  - ⇨ verstehen warum ein Blob kein guter Entwurf ist

Anschließend können Sie Vor- und Nachteile an einem Entwurf erkennen

Vorgehen im Projekt

- Situation im Projekt
  - ⇨ Ein relativ kleines Team von Systemanalytikern und Architekten hat die Anforderungen bestimmt und eine Architektur entworfen
  - ⇨ Die Architektur beschreibt das System auf einer abstrakten Ebene



- Verfeinere die SW-Architektur !
  - ⇨ Verteile die Komponenten, welche durch die Architektur festlegt sind, für die Ausarbeitung der Details
    - an interne Teams oder auch an Externe
  - ⇨ die Beziehungen zwischen den Komponenten legt die Architektur fest
    - Die Erfüllbarkeit der wesentlichen Anforderungen wurde mit der Architektur analysiert und "getestet"



Die SW-Architektur gibt einen Rahmen vor, der im Design ausgefüllt wird !



Die Aufgabe des Designs

- Die SW-Architektur
  - ⇨ gibt eine Zerlegung in Arbeitspakete mit Verantwortlichkeiten vor
  - ⇨ macht Vorgaben für das weitere Vorgehen und die Umsetzung (z.B. Installation, Prozesse, Verteilung auf Rechner usw.)
  - ⇨ Es fehlen noch sämtliche detaillierte Beschreibungen mit Methoden, Parametern, konkreten Abläufen
- Im Design
  - ⇨ werden Details ausgearbeitet
    - Klassen mit Methoden und Datentypen
    - Abläufe
    - Betriebssystemanbindung (Threads, Scheduler, Timer,…)
  - ⇨ entstehen Klassendiagramme, Sequenzdiagramme, Zustandsdiagramme usw.
  - ⇨ erfolgt die Umsetzung auf Programmiersprache(n), Betriebssystem(e), Hardware

⇨ Im Design wird das Konzept für die Implementierung der (einzelnen) Komponenten gemacht !



Gutes Design

- Im Design
  - ⇨ gibt es viele Lösungen
  - ⇨ sind viele Lösungen ungeeignet, einige sind geeignet
  - ⇨ je konkreter die Randbedingungen vorgegeben sind, desto weniger gute Lösungen gibt es – bis hin zu überhaupt keiner

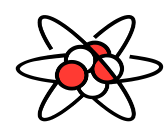
Frage: Welche Lösung ist gut, welche Lösung ist schlecht?

- Wege zu einem guten Design:
  - ⇨ Prinzipien lernen ⇨ Designprinzipien
  - ⇨ Regeln / Tipps und Tricks lernen (Best Practices) ⇨ Regeln
  - ⇨ aus eigenen Fehlern lernen ⇨ Üben
- Weitere Möglichkeiten
  - ⇨ Bekannte (und gute) Lösungen einsetzen ⇨ Design Patterns
  - ⇨ aus bekannten Fehlern lernen ⇨ Anti Patterns



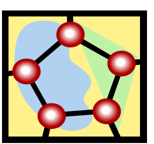
Zur Erinnerung aus OOAD: Regeln für ein gutes Design (I)

- Grundprinzipien
  - ⇨ Trennung von Zuständigkeiten => starker Zusammenhalt
    - jeder Teil / jede Klasse hat genau eine Aufgabe

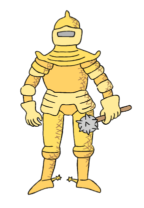




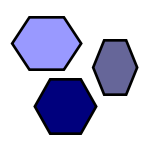
  - ⇨ Minimierung von Abhängigkeiten
    - Minimiere die Abhängigkeiten zwischen Systemteilen



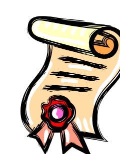
  - ⇨ Geheimnisprinzip
    - Kapsele das interne Wissen eines Systemteils und verrate es niemand anderem



  - ⇨ Homogenität
    - Löse ähnliche Probleme mit ähnlichen Lösungen und verwende ähnliche Strukturierungsgrößen innerhalb einer Strukturierungsebene



  - ⇨ Redundanzfreiheit
    - Keine Teile sind doppelt vorhanden





Zur Erinnerung aus OOAD: Regeln für ein gutes Design (II)

- Regeln zum Entwurf von Klassen
  - Setze Vererbung sparsam ein
  - Normalisiere das Datenmodell
  - Vermeide transitive Assoziationen
  - Analysiere m:n-Beziehungen genau
  - Vermeide 1:1-Beziehungen
  - Modelliere nur fachliche, nie technischen Aspekte
  - Beschränke Dich auf das für das System Notwendige
- Regeln zum Entwurf von Operationen und Schnittstellen
  - Operationen sind unabhängig von der verwendeten Technologie
  - Gib keine Referenzen nach außen
  - Normalisiere die Schnittstelle
  - Mache die Operationen grobgranular
  - Mache die Operationen idempotent
  - Mache die Operationen kontextfrei



Zur Erinnerung aus OOAD: Regeln für ein gutes Design (III)

- Es gibt weitere Prinzipien bzw. diverse Varianten dieser Prinzipien
  - die Kernaussage bzw. Zielrichtung ist allerdings seit vielen Jahren unumstritten
- Diese Grundprinzipien sind anwendbar
  - für implementierungsnahen Entwurf ("Feinentwurf")
  - aber auch für den Entwurf von Anwendungen auf höherer Abstraktionsebene ("Software Architektur") bis hin zu Anwendungslandschaften

Diese Regeln geben konkrete Leitlinien für den Entwurf !



Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

6.1 Wiederholung, für die, die nochmals nachschauen wollen:

Grundprinzipien für das Design



Idee für das 1. Grundprinzip



Trenne das, was nicht zusammen gehört –
oder andersrum:
Sorge dafür, dass (z. B. innerhalb einer Klasse) alles stark zusammenhängt !

- Für Funktionen, Klassen, Komponenten: "Starker Zusammenhalt" (Strong Cohesion)
  - ⇨ Kohäsion ist ein Maß dafür, wie eng die Beziehungen und der Fokus der Verantwortlichkeiten in einem Modul (Funktion, Klasse, ..) sind
    - Schwache Kohäsion:
      - Ein Modul ist verantwortlich für viele Aufgaben
    - Starke Kohäsion:
      - Jeder Modul hat einen einzigen, klaren Zweck

        z. B. jede Operation einer Klasse hat einen einzigen, klaren Zweck und

        z. B. die Operationen einer Klasse bilden eine Gruppe zusammengehörender Aufgaben



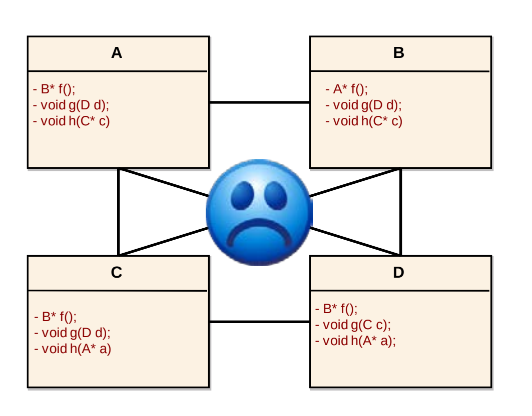
Grundprinzip 1: Starker Zusammenhalt (Strong Cohesion), Trennung von Zuständigkeiten (Separation of Concerns)

- Unterteile das System so in Teile, dass jeder Teil genau eine Aufgabe hat !
  - ⇨ Diese Aufgabe wird auch nur von diesem Teil erledigt
  - ⇨ Die Abgrenzung gegenüber anderen Teilen ist klar erkennbar
  - ⇨ Die Aufgabe ist genau und prägnant definiert und spiegelt sich im Namen des Teils wider
  - ⇨ Trenne Teile an denen sich (voraussichtlich) keine Änderungen ergeben, von Teilen an denen sich wahrscheinlich Änderungen ergeben!
- Vorteile von getrennten Zuständigkeiten:
  - ⇨ Ein Systemteil, der nur eine Zuständigkeit hat, ändert sich nur, wenn an dieser einen Zuständigkeit eine Änderung nötig wird!

⇨ Trennung von Zuständigkeiten fördert die Wartbarkeit (Gute Verständlichkeit), Erweiterbarkeit und Wiederverwendbarkeit !



Idee für das 2. Grundprinzip

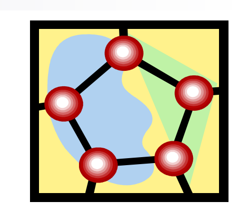


Ein System ist schwer zu handhaben, wenn jeder jeden kennt!
Reduziere die Anzahl der Beziehungen auf das Nötige!

- Für Funktionen, Klassen, Komponenten: "Schwache Kopplung" (Loose Coupling)
  - ⇨ Kopplung ist ein Maß dafür, wie sehr ein Element von anderen abhängt
  - ⇨ z. B. Klasse A ist von B abhängig, wenn:
    - A hat ein Attribut vom Typ B oder verwendet eine Methode eines Objekts vom Typ B
    - A hat eine Methode mit Parameter/Return vom Typ B
    - A ist von B abgeleitet oder A implementiert die Schnittstelle von B
  - ⇨ Das Prinzip der schwachen Kopplung fordert Klassen so zu definieren, dass nur die nötige Kopplung da ist



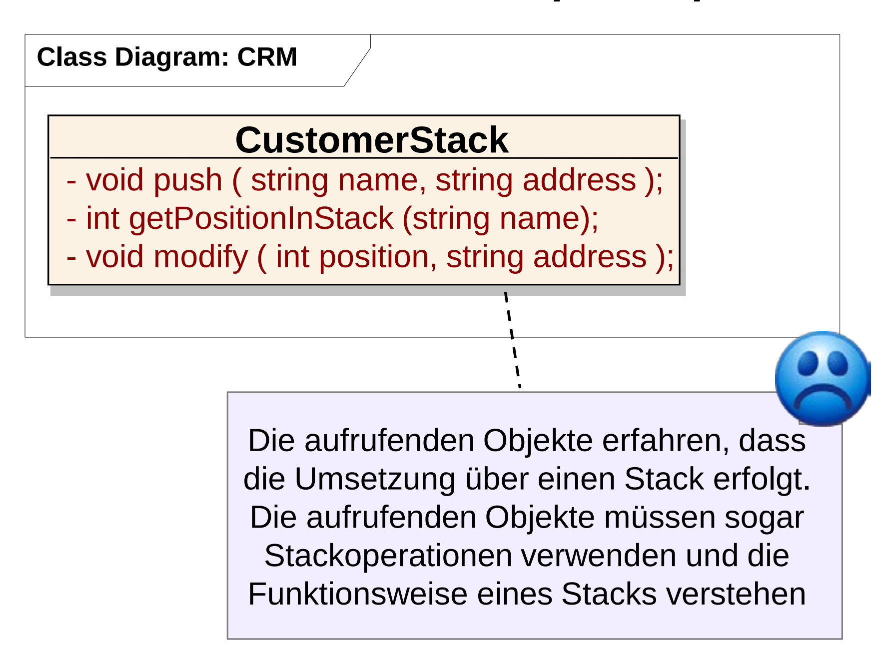
Grundprinzip 2: Schwache Kopplung (Loose Coupling) Minimierung von Abhängigkeiten



- Minimiere die Abhängigkeiten zwischen Systemteilen !
  - ⇨ fasse stark zusammenhängende Teile zusammen
  - ⇨ überprüfe die Verantwortlichkeiten auf eine saubere Trennung
  - ⇨ entkopple abhängige Teile (evtl. durch Auslagerung in einen neuen Teil)
- Vorteile von minimierten Abhängigkeiten:
  - ⇨ Veränderungen eines Systemteils tangieren nur wenige andere Klassen !

⇨ Minimierung von Abhängigkeiten fördert die Verständlichkeit, Wiederverwendbarkeit und Wartbarkeit !

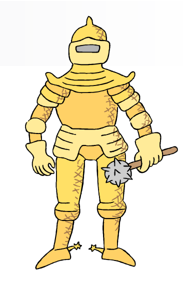
Idee für das 3. Grundprinzip



Änderungen werden erschwert, wenn die interne Realisierung eines Systemteils anderen Systemteilen bekannt ist und diese die Interna ansprechen und verwenden !

- Eselsbrücke "Geheimagenten-Prinzip":
  - ⇨ Jeder "verrät" nur das, was ein anderer wissen muss, um mit ihm zu arbeiten!
  - ⇨ Jeder "weiß" nur das, was er wissen muss, um seinen Job zu machen!

Grundprinzip 3: Geheimnisprinzip ("Information Hiding")

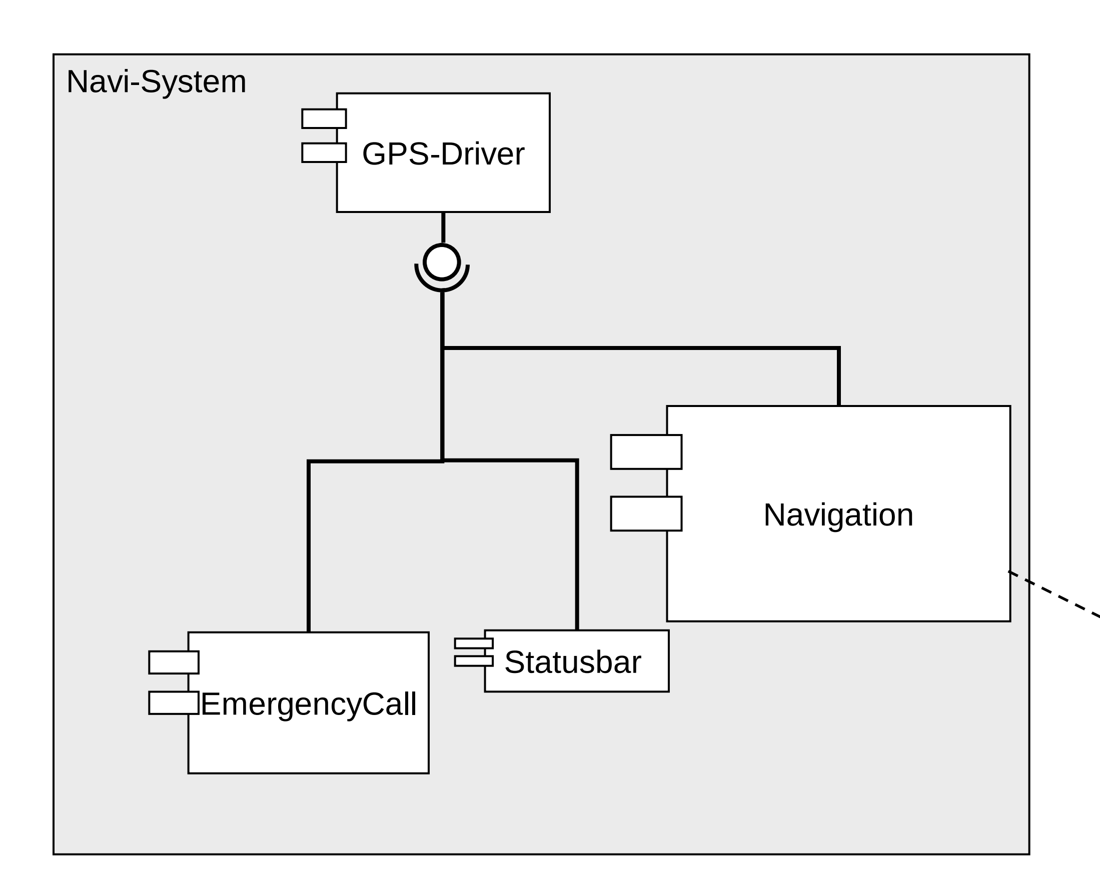


- Kapsele das interne Wissen eines Systemteils und verrate es niemand anderem!
  - ⇨ Abläufe, Daten und Strukturen sind gekapselt
  - ⇨ Systemteile sind benutzbar ohne Kenntnis der Realisierung (Nur Schnittstelle ist bekannt, nicht die Implementierung)
  - ⇨ interne Änderungen sind von außen nicht sichtbar
- Umsetzung in C++:
  - ⇨ Sichtbarkeit "private" für alle Attribute und interne Methoden
- Vorteile des Geheimnisprinzips:
  - ⇨ unterstützt Trennung von Zuständigkeiten und Minimierung der Abhängigkeiten
  - ⇨ erleichtert verteilte Entwicklung

⇨ Das Geheimnisprinzip fördert die
Wartbarkeit, Änderbarkeit und Testbarkeit !



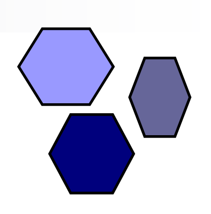
Idee für das 4. Grundprinzip



Ein uneinheitlicher Entwurf erschwert das Verständnis !

Die Komponenten des Systems haben stark unterschiedliche Komplexität (z.B. die gesamte Navigation im Vergleich zu einem Statusbar) und sind auch nicht intuitiv angeordnet (z.B. in Schichten für Treiber, Anwendungen usw.))

Grundprinzip 4: Homogenität



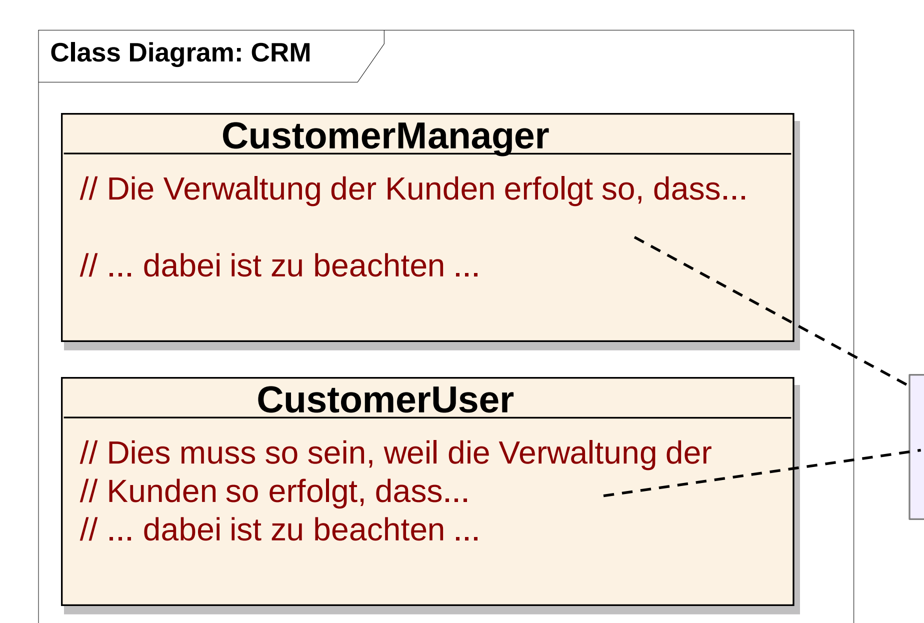
- Verwende ähnliche Strukturierungsgrößen innerhalb einer Strukturierungsebene
  - ⇨ entwerfe Systemteile innerhalb einer Strukturierungsebene (z.B. Anwendungsschicht) so, dass die Größe und Komplexität ungefähr gleich ist
  - ⇨ verwende bei Bedarf mehrere - jeweils homogene - Strukturierungsebenen
- löse ähnliche Probleme mit ähnlichen Lösungen!
  - ⇨ verwende bereits bekannte Lösungen erneut – sofern es keine wichtigen Gründe für eine andere Lösung gibt
- Vorteile der Homogenität:
  - ⇨ vermeidet unnötiges Neuerfinden von vorhandenen Lösungen
  - ⇨ erleichtert die Entscheidungsfindung

> ⇨ Homogenität fördert die
> Wartbarkeit und die Verständlichkeit !



Idee für das 5. Grundprinzip

Know-how (in jeglicher Form), das an mehreren Stellen im Entwurf auftaucht, wird früher oder später inkonsistent und führt zu Problemen !



Die spezielle Art der Kundenverwaltung wird an mehreren Stellen berücksichtigt



Grundprinzip 5: Redundanzfreiheit (DRY - Don't Repeat Yourself)

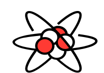
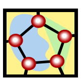
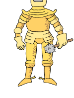
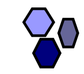


- Entwerfe das System so, dass jedes Stück Know-how in dem System nur an genau einer Stelle umgesetzt wird, die eindeutig und zuständig ist !
  - ⇨ Damit ist nicht nur Code gemeint! Es kann sich um alle Teile handeln, in denen Know-how steckt – also auch um Teile des Entwurfs, ein Datenbankschema oder auch die Dokumentation
  - ⇨ ziehe mehrfach verwendete Teil heraus und mache Sie für Andere verfügbar
  - ⇨ finde eine eindeutige und sinnvolle "Heimat" für das Know-how
  - ⇨ vermeide "Copy & Paste – Entwicklung"
- Vorteile der Redundanzfreiheit:
  - ⇨ fördert die Trennung der Zuständigkeiten (man muss sich entscheiden)
  - ⇨ Korrekturen müssen nur noch an einer Stelle erfolgen

⇨ Redundanzfreiheit fördert die Wiederverwendbarkeit, Wartbarkeit und die Verständlichkeit !



Zusammenfassung der Grundprinzipien

- 5 Grundprinzipien
  - ⇨ Trennung von Zuständigkeiten (Kohäsion / Zusammenhalt) 
  - ⇨ Minimierung von Abhängigkeiten (schwache Kopplung) 
  - ⇨ Geheimnisprinzip 
  - ⇨ Homogenität 
  - ⇨ Redundanzfreiheit 
- Es gibt weitere Prinzipien bzw. diverse Varianten dieser Prinzipien
  - ⇨ die Kernaussage bzw. Zielrichtung ist allerdings seit vielen Jahren unumstritten
- Diese Grundprinzipien sind anwendbar
  - ⇨ für implementierungsnahen Entwurf ("Feinentwurf")
  - ⇨ aber auch für den Entwurf von Anwendungen auf höherer Abstraktionsebene ("Software Architektur") bis hin zu Anwendungslandschaften



Hochschule Darmstadt
Fachbereich Informatik

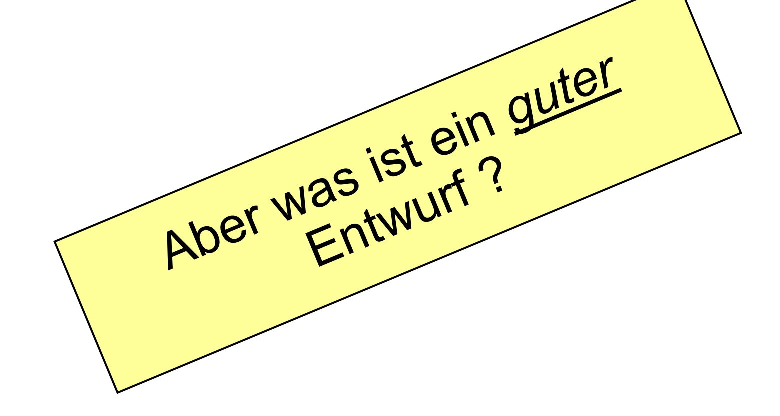
Objektorientierte Analyse und Design

6.2 Wiederholung, für die, die nochmals nachschauen wollen:

Regeln zum Entwurf von Klassen



Zur Erinnerung: Schritte zum Entwickeln von Klassendiagrammen



1. Klassen(kandidaten) finden
   - ⇨ Substantive bestimmen
   - ⇨ überflüssige Begriffe rausfiltern
   - ⇨ Attribute identifizieren
   - ⇨ Operationen über Verben suchen
2. Spezialisierungsbeziehungen suchen
   - ⇨ Ähnliche Klassen identifizieren (Aber: "Ist-ein-Regel" beachten!)
   - ⇨ Evtl. Schnittstellen durch abstrakte Klasse + Spezialisierung definieren
3. Assoziationen zwischen Klassen bestimmen
   - ⇨ "Kennt"-Beziehungen: normale Assoziation
   - ⇨ "Besteht aus"-Beziehung: Aggregation bzw. Komposition
   - ⇨ evtl. Leserichtung und Rollen angeben
   - ⇨ Objekte, deren Existenz an einer Assoziation hängt: Assoziationsklasse
4. Multiplizitäten und Navigationsrichtungen eintragen



Entwurf von Klassen

- Bisher kennen Sie ein "Standardverfahren" zur Bestimmung von Klassen und den Beziehungen dazwischen
  - ⇨ es geht darum, die Klassen so zu entwerfen, dass die Qualitätsanforderungen möglichst gut erfüllt werden
  - ⇨ aber es gibt immer viele verschiedene Lösungen mit unterschiedlichen Eigenschaften, welche die Qualitätseigenschaften mehr oder weniger gut erfüllen
- Wir betrachten nun einige Beispiele und
  - ⇨ erarbeiten jeweils eine (naheliegende) Lösung
  - ⇨ analysieren die Nachteile dieser Lösung
  - ⇨ diskutieren eine zweite Lösung, welche die Nachteile nicht hat
  - ⇨ leiten daraus eine Regel für den Entwurf von Klassen ab

Aufgabe 1

- Modellieren Sie folgenden Sachverhalt:
  - ⇨ Ein Softwaresystem für eine Firma soll Kunden, Lieferanten und Mitarbeiter verwalten
  - ⇨ Alle werden durch Name und Adresse identifiziert
  - ⇨ Ein Mitarbeiter ist entweder Arbeiter oder Manager

- Prüfen Sie, wie die folgenden naheliegenden Punkte mit Ihrer Lösung umgesetzt werden können:
  - ⇨ Kunden können gleichzeitig auch Lieferanten sein
  - ⇨ Arbeiter können zu Managern befördert werden
  - ⇨ In Zukunft können auch weitere Personen wie Gesellschafter etc., aber auch weitere Laufbahnstufen wie Ingenieur etc. relevant werden



Lösung A: mit Generalisierung / Spezialisierung

Class Diagramm Person mit Vererbung

Person
- adresse: String
- name: String

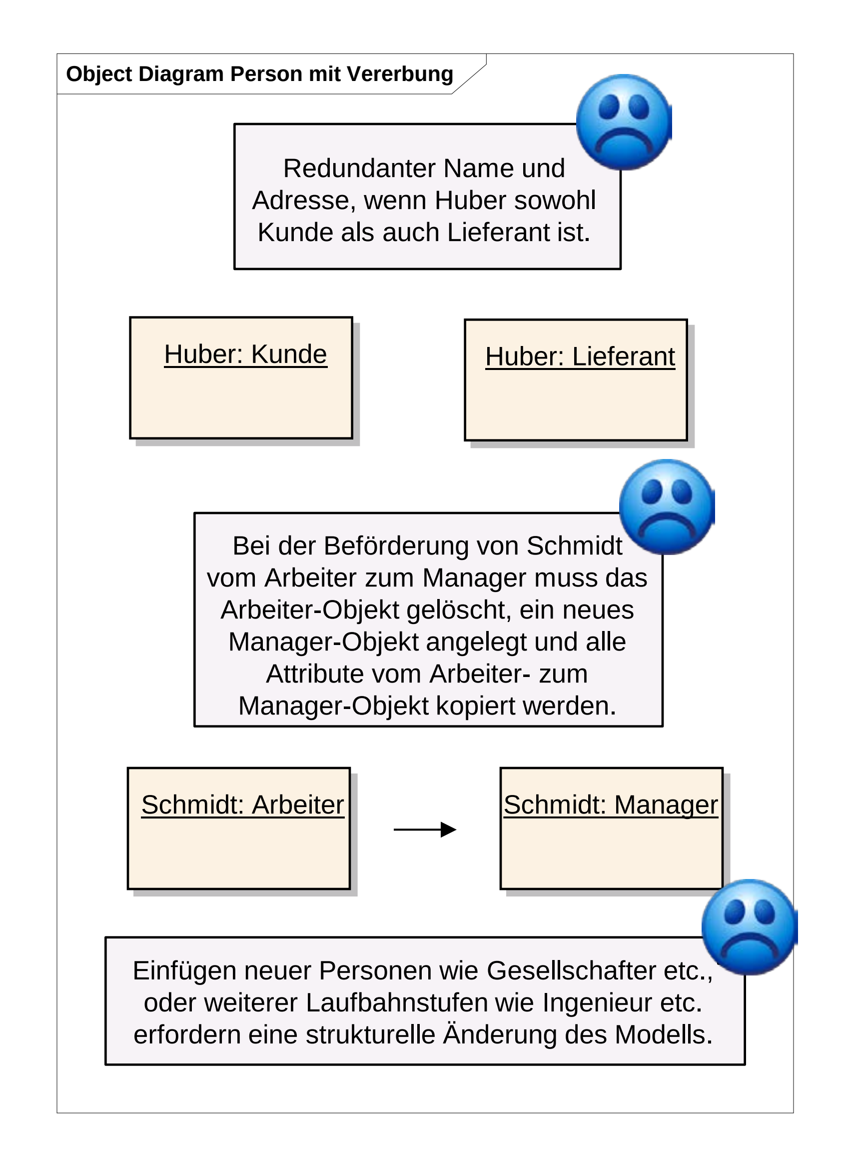
Kunde

Lieferant

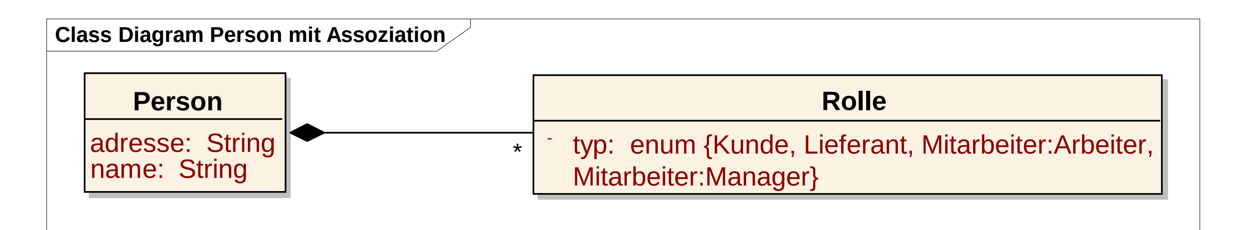
Mitarbeiter

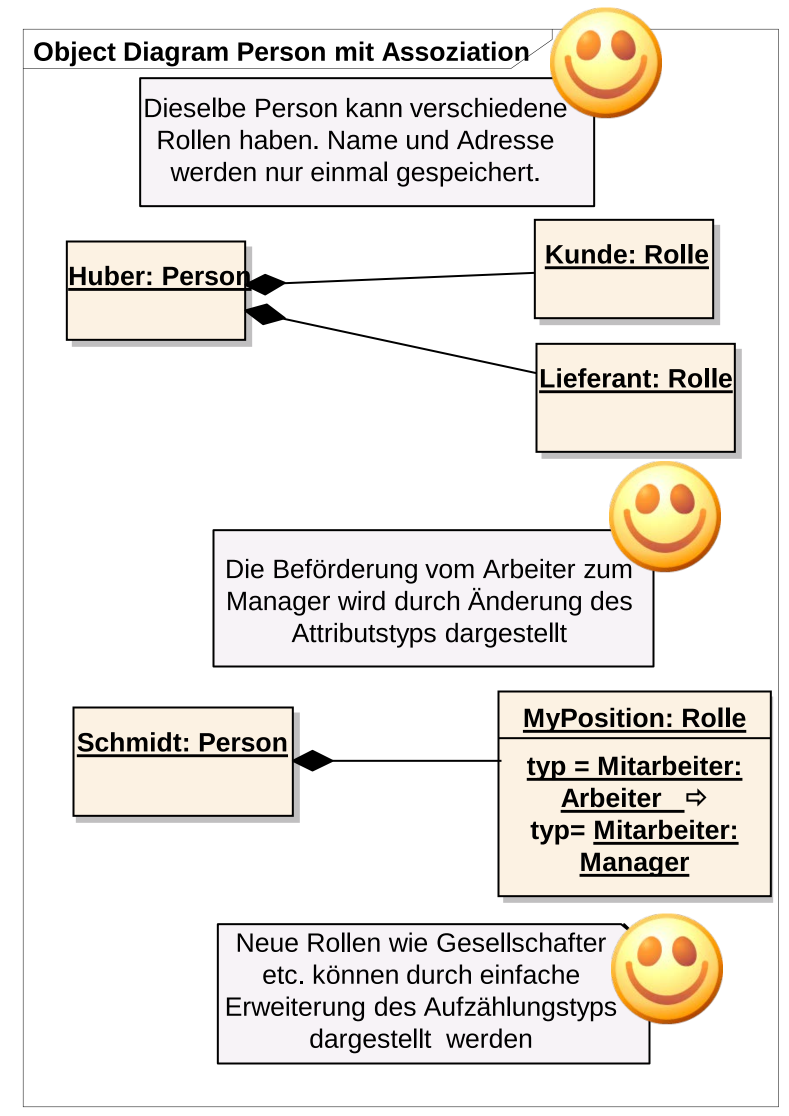
Arbeiter

Manager



Lösung B: Mit Assoziation (bzw. Komposition, Aggregation)





Warum nicht als Attribut Rolle[ ] in Person?

Regel 1: Setze Vererbung (Generalisierung / Spezialisierung) sparsam ein

- Vererbung (Generalisierung / Spezialisierung) ist das am meisten überschätzte und überbenutzte Konzept der Objekt-Orientierung
  - ⇨ Setze Vererbung nur bei echter „ist ein“ Beziehung ein
  - ⇨ Ersetze, wo sinnvoll möglich, eine Vererbung durch eine Assoziation
- Beispiele:
  - ⇨ Ein Kunde ist eine Person → eine Person hat die Rolle Kunde
  - ⇨ Eine Frau ist eine Person → eine Person hat das Geschlecht weiblich
  - ⇨ Ein Rothaariger ist eine Person → eine Person hat die Haarfarbe rot
- Verwende nur einfache Vererbung (Single Inheritance)
  - ⇨ „Bei der Einfachvererbung hat man nur einen Schuss – und der muss sitzen!“
- Schachtele Vererbungsbäume nicht zu tief
  - ⇨ Tiefe 2-3 reicht meistens aus

⇨ Sparsame Verwendung von Vererbung fördert die Einfachheit und Erweiterbarkeit von Modellen !



Aufgabe 2

- Modellieren Sie folgenden Sachverhalt:
  - ⇨ In einem MP3-Archiv sollen Lieder gespeichert werden
  - ⇨ Für jedes Lied soll der Titel des Lieds, Titellänge, der Interpret, der Wohnort des Interpreten, der Name des Albums und das Bild des Album-Covers gespeichert werden

- Prüfen Sie, wie die folgenden naheliegenden Punkte mit Ihrer Lösung umgesetzt werden können:
  - ⇨ Wurde der Name des Interpreten oder des Albums falsch geschrieben, so kann dies leicht korrigiert werden
  - ⇨ In der nächsten Version sollen zu einem Album weitere Daten gespeichert werden wie z.B. das Genre (Klassik, Rock, ...)

Lösung A: mit Redundanzen

Class Diagram Lied

Lied

interpret: String
intepret-wohnort: String
album: String
album-cover: Bild
titel: String
titel-länge: Int

Object Diagram Lied

Not That Kind: Lied

interpret = Anastacia ...
album = Not That Kind ...
...

I'm Outta Love: Lied

interpret = Anastacia ...
album = Not That Kind ...
...

Cowboys & Kisses: Lied

interpret = Anastacia ...
album = Not That Kind ...
...

Shine On You Crazy Diamond: Lied

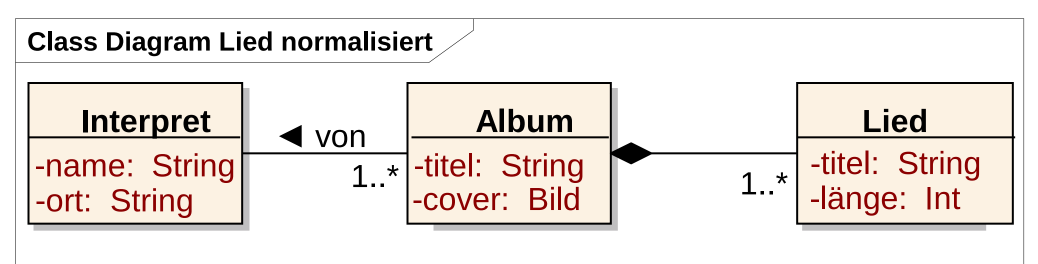
interpret = Pink Floyd ...
album = Wish You Were Here ...
...

Ist der Name des Interpreten oder das Album falsch geschrieben, so muss dies in allen Objekten vom Typ Lied korrigiert werden. Inkonsistente Einträge können leicht entstehen.

Weitere zu Album gehörende Daten wie z.B. das Album-Cover sind in allen Liedern redundant vorhanden.

⇨ Die Normalisierung das Datenmodells reduziert Redundanzen !

Lösung B: normalisiert



Object Diagram Lied normalisiert

Not That Kind: Lied

Anastacia: Interpret — von — Not That Kind: Album — I'm Outta Love: Lied

Cowboys & Kisses: Lied

Pink Floyd: Interpret — von — Wish You Were Here: Album — Shine On You Crazy Diamond :Lied

Normalisierung reduziert Redundanzen:

siehe Vorlesung Datenbanken!

Ist der Name des Interpreten oder das Album falsch geschrieben, so muss dies nur im Objekt Interpret bzw. Album korrigiert werden.

Weitere zu Album gehörende Daten wie z.B. das Album-Cover sind nicht redundant vorhanden.

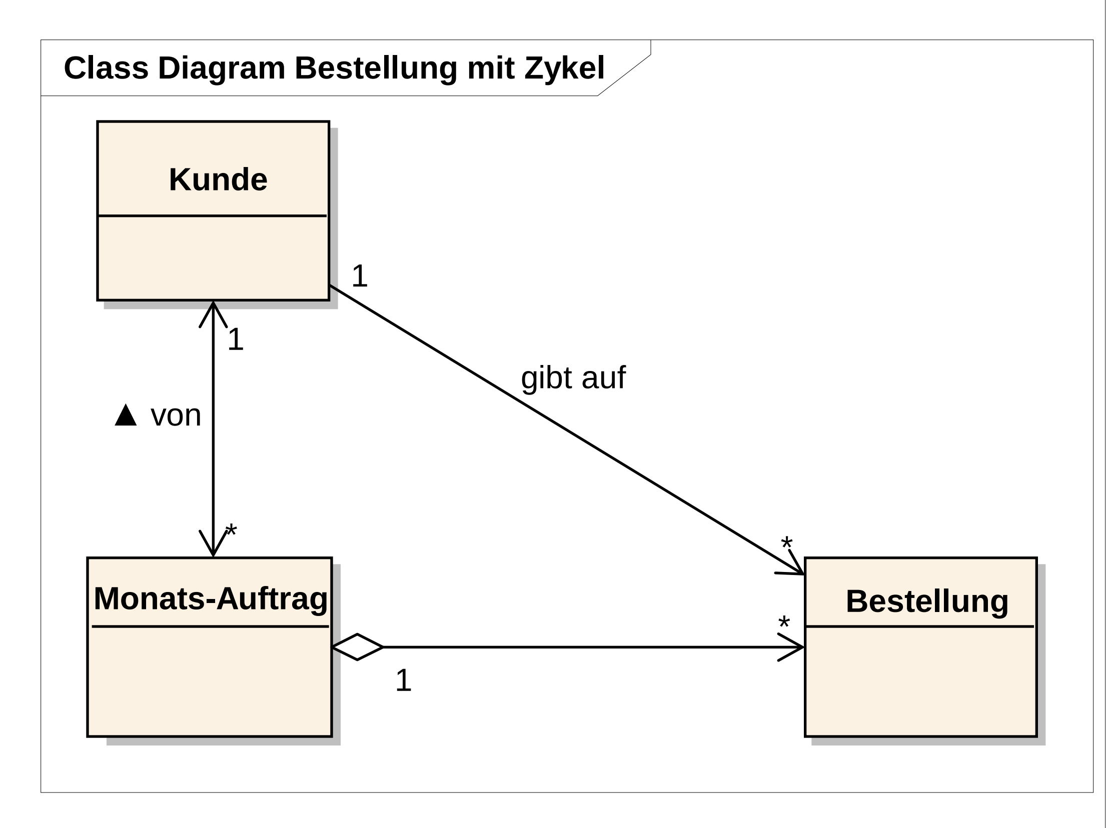


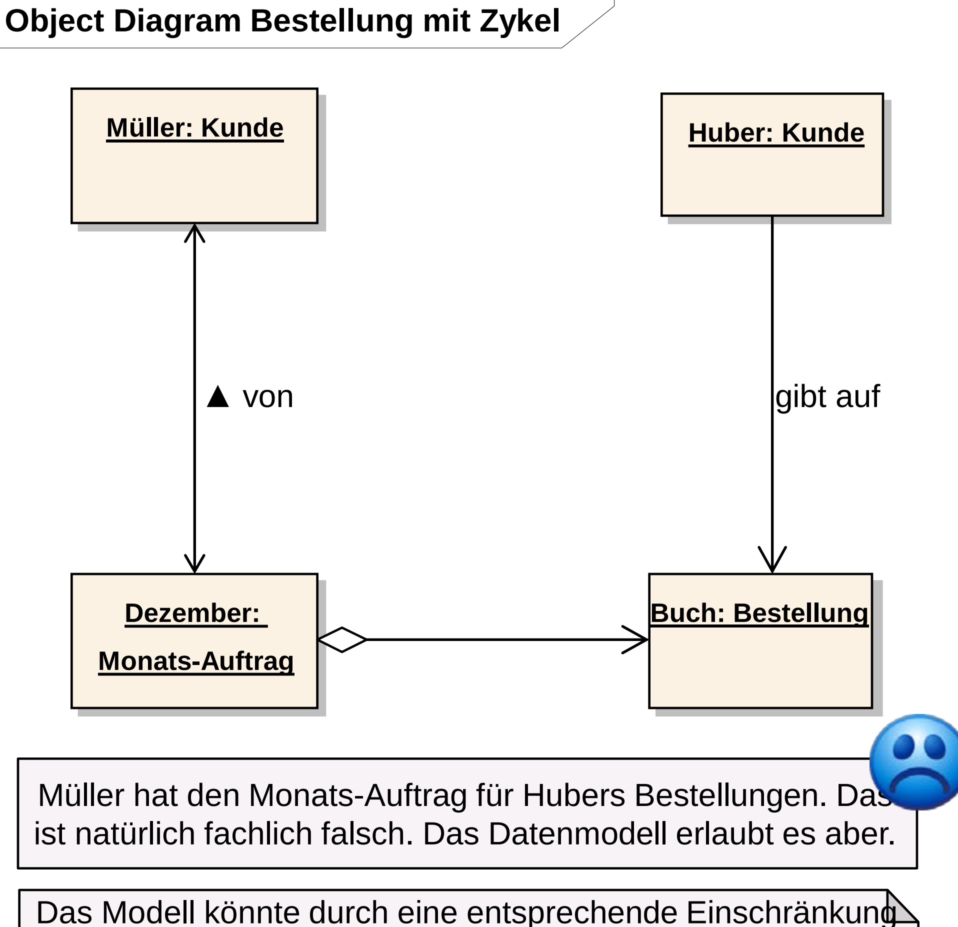
Aufgabe 3

- Modellieren Sie folgenden Sachverhalt aus einem Auftragssystem:
  - ⇨ Kunden können Bestellungen aufgeben
  - ⇨ Mehrere Bestellungen zusammen bilden einen Monats-Auftrag
  - ⇨ Ein Monats-Auftrag bezieht sich immer auf einen Kunden



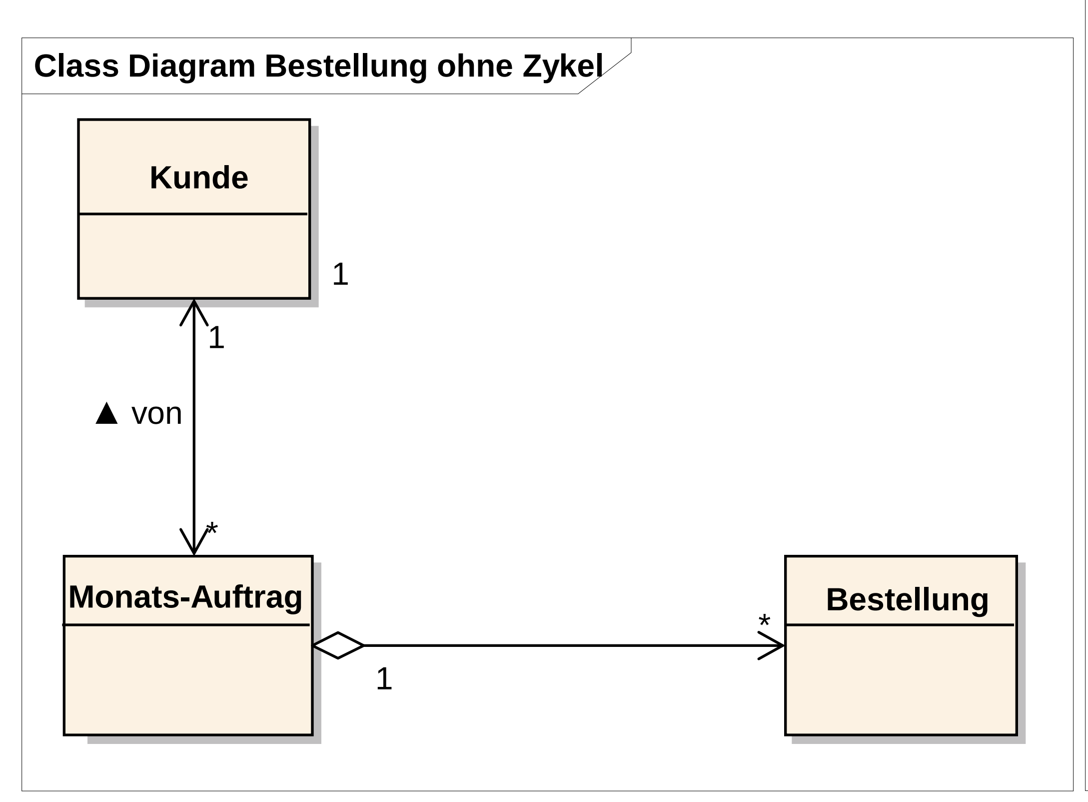
Lösung A: Transitive Assoziationen

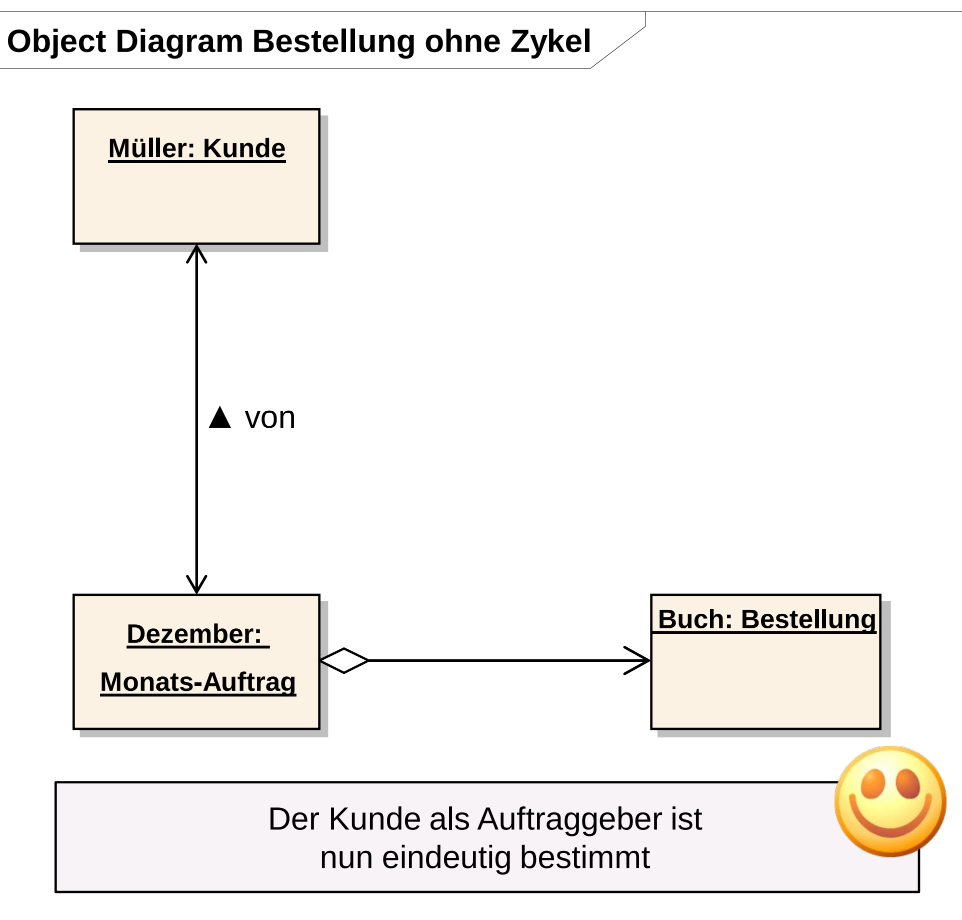






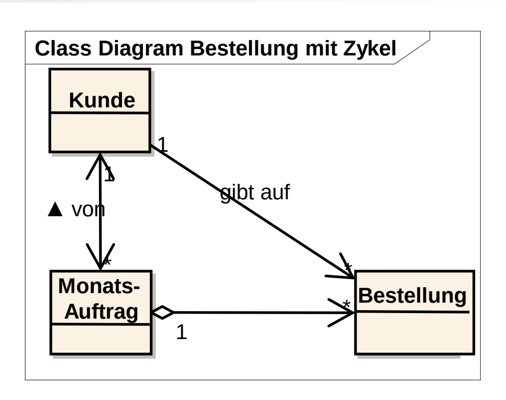
Lösung B: Ohne transitive Assoziationen







Regel 3: Vermeide transitive Assoziationen



- Transitive Beziehungen bieten mehrere Wege für eine Beziehung zwischen zwei Objekten
  - ⇨ in Lösung A gibt es
    - die direkte Beziehung "gibt auf" von Kunde zu Bestellung
    - eine Beziehung von Kunde über Auftrag zu Bestellung
  - ⇨ Man kann die Beziehungen mit unterschiedlichen Objekten assoziieren (wie in Lösung A)
- Vermeide transitive Assoziationen
  - ⇨ Transitive Beziehungen können meist durch Weglassen von (redundanten) Assoziationen aufgelöst werden
  - ⇨ Falls transitive Beziehungen nicht vermeidbar sind, müssen Widersprüche über Constraints ausgeschlossen werden
    z.B. Kunde der Bestellung muss gleich sein dem Kunden des Auftrags

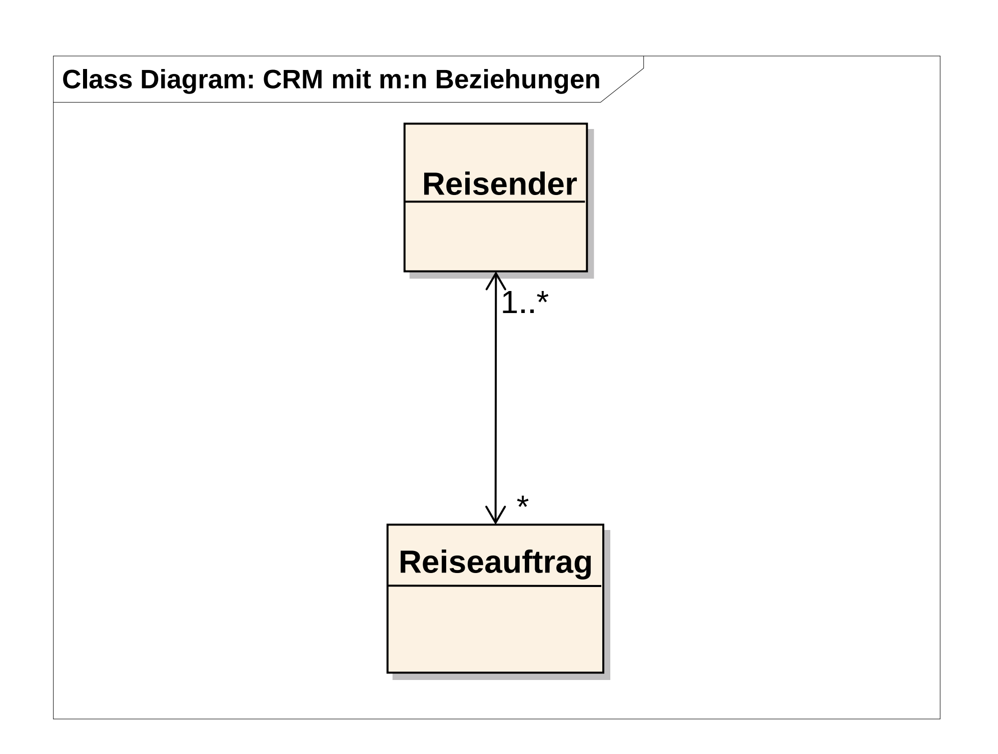
⇨ Vermeidung von transitiven Beziehungen reduziert Redundanzen und erhöht die Korrektheit!



Aufgabe 4

- Es soll ein Customer Relation Management (CRM) für einen Reiseveranstalter entworfen werden. Modellieren Sie folgenden Sachverhalt:
  - ⇨ Reisende können viele Reisen buchen (d.h. Reiseaufträge erteilen)
  - ⇨ Ein Reiseauftrag kann mehrere Reisende umfassen
- Prüfen Sie, wie der folgende naheliegende Punkt mit Ihrer Lösung umgesetzt werden kann:
  - ⇨ Das System soll in der Wartungsphase so umgebaut werden, dass zusätzlich der Status eines Visa für jede Reise und jeden Reisenden erfasst werden kann

Lösung A: m:n-Beziehungen



Object Diagram: CRM mit m:n Beziehungen

Huber: Kunde

Mallorca: Reiseauftrag

Indien: Reiseauftrag

Mayer: Kunde

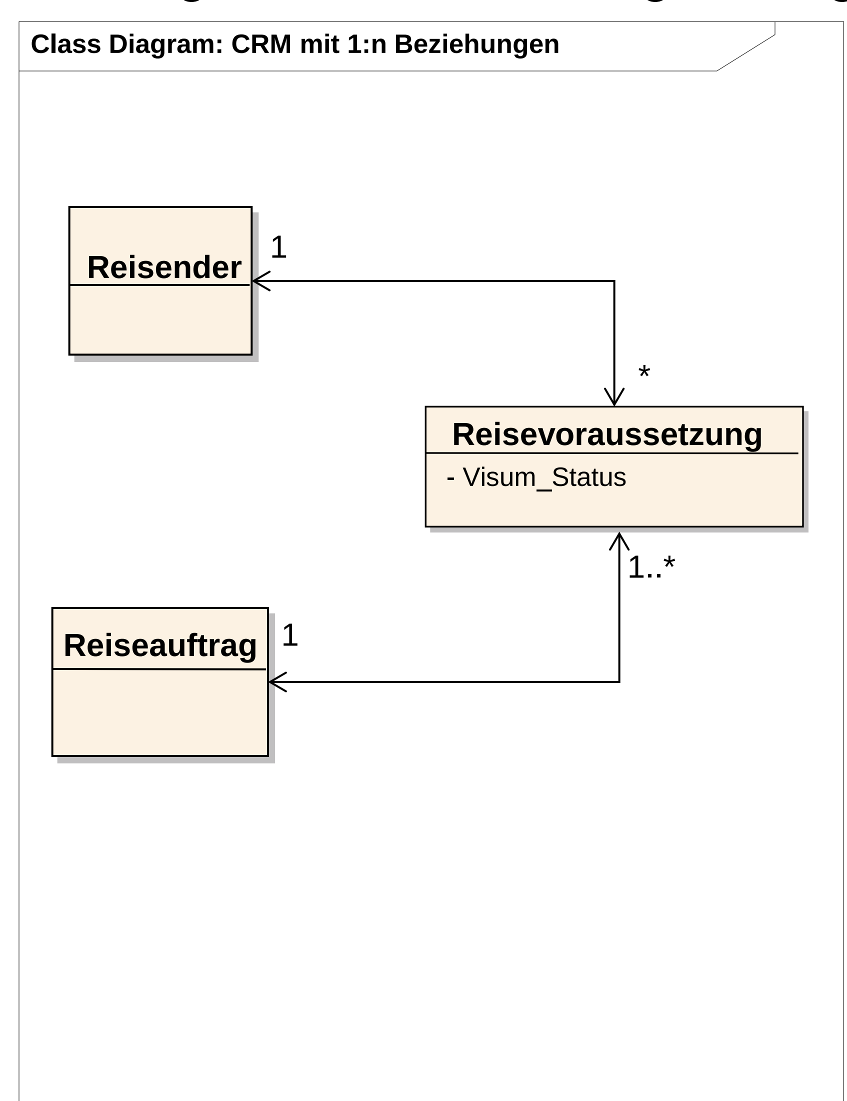
Russland: Reiseauftrag

HuberNachIndien: Reisevoraussetzung

Ein Visum ist abhängig vom Reisenden und vom Reiseziel. Der Zustand muss in einer Assoziationsklasse abgespeichert werden!
Das erfordert allerdings in der Realisierung eine neue Klasse und Änderungen an Reisender oder Reiseauftrag!



Lösung B: m:n-Beziehungen aufgelöst



Object Diagram: CRM mit 1:n Beziehungen

Huber: Reisender

HuberNachMallorca: Reisevoraussetzung
- Visum: keins

Mallorca: Reiseauftrag

HuberNachIndien: Reisevoraussetzung
- Visum: OK

Indien: Reiseauftrag

MayerNachIndien: Reisevoraussetzung
- Visum: beantragt

Mayer: Reisender

MayerNachRussland: Reisevoraussetzung
- Visum: OK

Russland: Reiseauftrag

Zu einem Reiseauftrag können nun für alle Reisenden die Reisevoraussetzungen gespeichert werden.
Damit keine transitive Assoziation entsteht, wird die Assoziation zwischen Reiseauftrag und Reisender jetzt über die Reisevoraussetzung geführt.

Regel 4: Analysiere m:n-Beziehungen genau

- Hinter m:n-Beziehungen können sich leicht Probleme verbergen
  - ⇨ Hinter m:n-Beziehungen stecken oft eigenständige Klassen (vgl. Assoziations-klassen) oder auch Attribute, die in der Analyse nicht aufgetaucht sind
    - entdeckt man solche Klassen oder Attribute erst in der Wartungsphase, entsteht unerwünschter Aufwand, weil die Änderung mehrere Klassen betrifft
  - ⇨ Das gezielte Hinterfragen der m:n-Beziehung löst im Beispiel das Problem:
    - "Gibt es irgendetwas das jeder einzelne Reisende für eine Reise braucht?" liefert "Ja, für diverse Länder braucht man ein spezielles Visum" (⇨ Reisevoraussetzung)
- Wandle m:n-Beziehungen um, wenn es sinnvoll erscheint
  - ⇨ Löse m:n-Beziehungen in (mehrere) 1:n-Beziehungen auf
  - ⇨ Füge dazu neue Klassen ein (vgl. Umsetzung von Assoziationsklassen in C++) und finde fachlich passende Begriffe für die neuen Entitäten.
    - Manchmal gibt es noch keinen Begriff für die neue Entität. Dieser sollte dann zusammen mit der Fachabteilung festgelegt werden
    - Häufig enthalten die neu entstehenden Entitäten weitere fachliche Informationen (z.B. ein Datum).

Weitere Regeln - eigentlich selbstverständlich, aber ...

- Vermeide 1:1-Beziehungen
  - ⇨ Stehen zwei Entitäten in einer 1:1-Beziehung, so sollte man sie - sofern fachlich sinnvoll - in eine Entität zusammenfassen. Das vereinfacht das Modell. Beispiel: Auftrag und Auftragskopf → Auftrag

- Modelliere nur fachliche, nie technischen Aspekte
  - ⇨ Beispiel: Eine Entität "Liste" oder gar "VerketteteListe" gibt es nicht (= Realisierung von x:n-Assoziation)

- Beschränke Dich auf das für das System Notwendige
  - ⇨ Beispiel: Sollen für einen Reisekatalog die Ausstattungen von Hotels (Pool, Sauna, Tennis, etc.) modelliert werden, so sind nur die Kategorien relevant. Irrelevant ist hier, dass Tennis eine Sportart ist, ob Pool und Saunabereich aneinander grenzen etc.



Zusammenfassung der Regeln

- Regeln zum Entwurf von Klassen
  - ⇨ Setze Vererbung sparsam ein
  - ⇨ Normalisiere das Datenmodell
  - ⇨ Vermeide transitive Assoziationen
  - ⇨ Analysiere m:n-Beziehungen genau
  - ⇨ Vermeide 1:1-Beziehungen
  - ⇨ Modelliere nur fachliche, nie technischen Aspekte
  - ⇨ Beschränke Dich auf das für das System Notwendige

⇨ Diese Regeln geben konkrete Leitlinien zum Entwurf von Klassen und deren Beziehungen !

Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

6.3 Entwurfsmuster



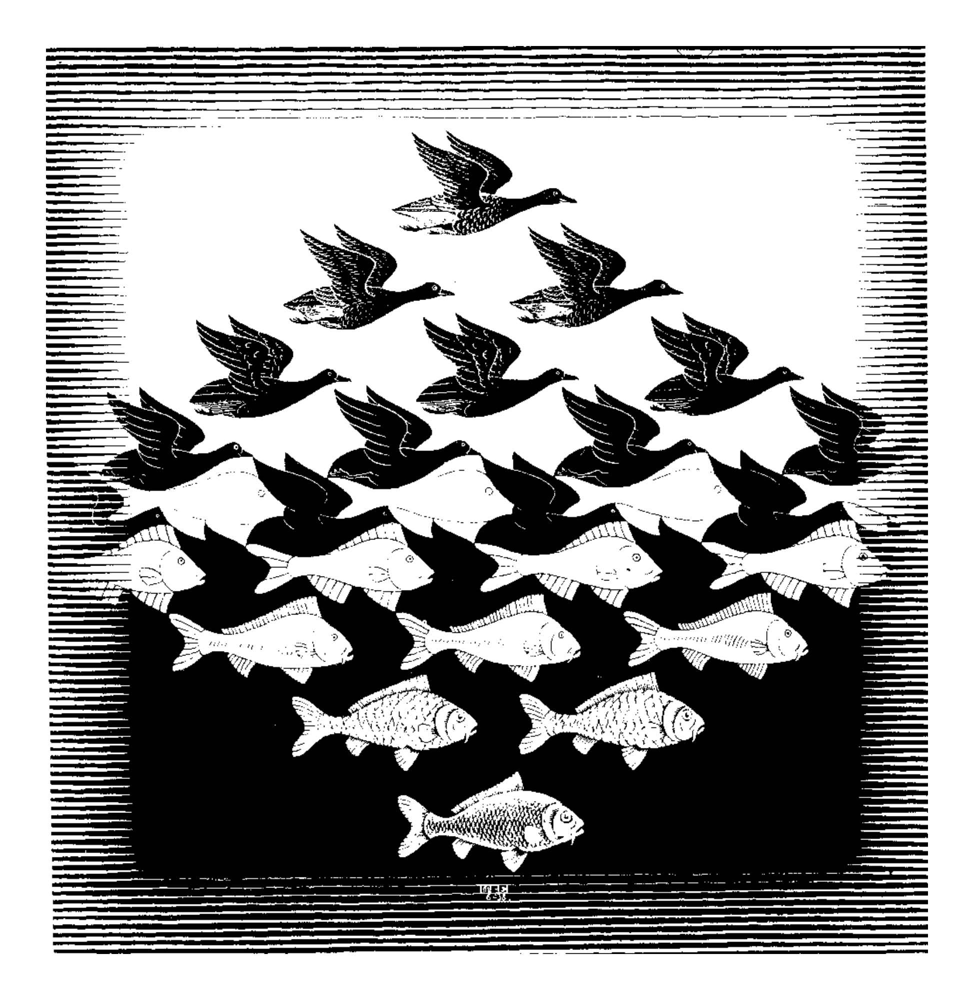
Muster (Patterns)

„Auf der Grundlage von Erfahrung und der Reflexion darüber...

...können wir wiederkehrende

Muster erkennen ...

Einmal erkannte Muster leiten unsere Wahrnehmung.“



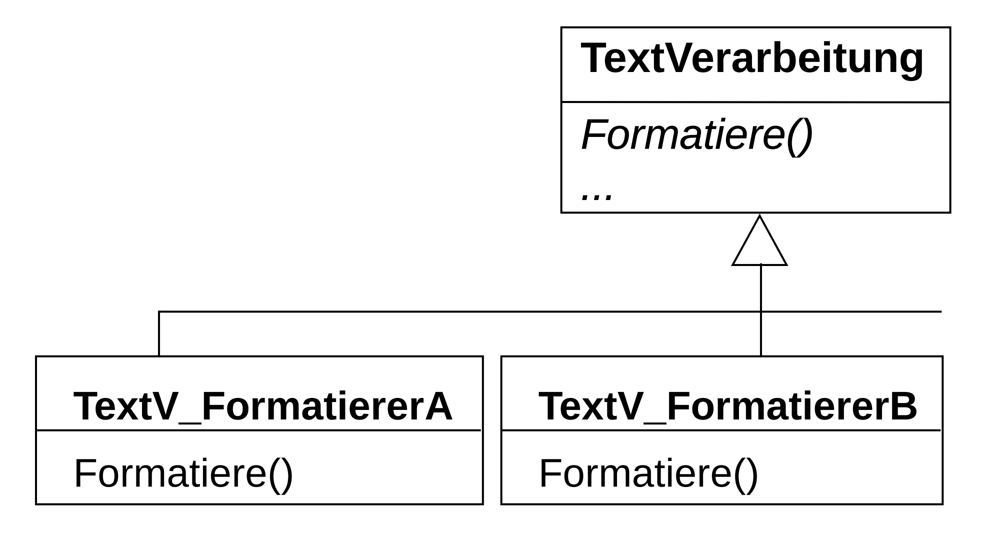
M.C. Escher: Luft und Wasser 8

Ein Problem aus der IT Welt

- Beispiel: Absatzformatierung in einer Textverarbeitung
  - ⇨ es gibt verschiedene Strategien um Absätze schnell oder schön zu formatieren
  - ⇨ neue Strategien sollen leicht einbaubar sein (am besten ohne Änderung in der eigentlichen Textverarbeitung)

1. Idee: Implementiere die "Textverarbeitung" als eine Klasse mit einer abstrakten Funktion "Formatiere()" und realisiere diese dann durch Spezialisierung!





- ⇨ dann entsteht eine Klasse für jede Strategie
  - wenn aber z.B. die Rechtschreibkorrektur ebenfalls in verschiedenen Verfahren existiert, dann gäbe es für alle Kombinationen von Formatierern und Korrekturen je eine Klasse z.B. TextV_FormatiererA_KorrekturA, …

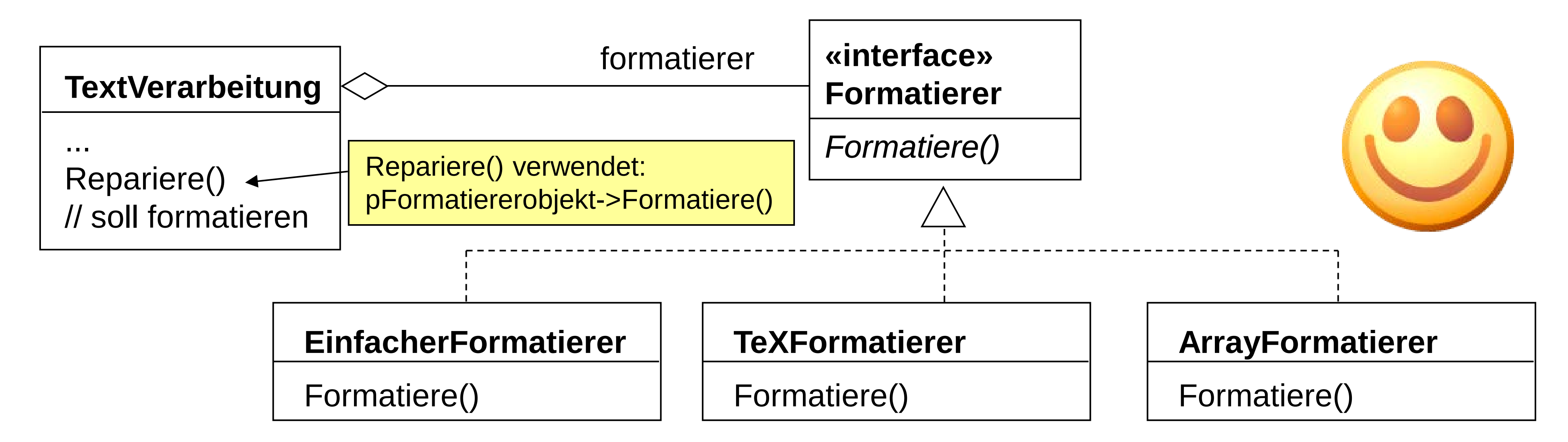


Ein Problem aus der IT Welt (2. Versuch)

2. Idee: Die Textverarbeitung muss gar nicht "wissen", welche Formatierung (oder Rechtschreibkorrektur) gewünscht ist – es reicht, wenn die Aufrufweise bekannt ist!

- ⇨ Definiere eine einheitliche Schnittstelle für die austauschbaren Algorithmen
  - ein Interface mit einer Methode (z.B. *Formatiere* oder *Rechtsschreibkorrektur*)
- ⇨ Jeder Algorithmus erbt diese Schnittstelle und implementiert sie
- ⇨ Der gewünschte Algorithmus muss der Textverarbeitung bekannt sein
  - Realisierung als Aggregation (Pointer auf konkreten Formatierer)





Frage: Wie kann hier Rechtschreibkorrektur integriert werden?



Das Strategie-Muster

Struktur

Kontext
KontextSchnittstelle()

strategie

«interface»
Strategie
AlgorithmusSchnittstelle()

Schnittstelle definiert eine Operation

KonkreteStrategieA
AlgorithmusSchnittstelle()

KonkreteStrategieB
AlgorithmusSchnittstelle()

KonkreteStrategieC
AlgorithmusSchnittstelle()

- Teilnehmer und Interaktionen:
  - **Kontext:** Wird mit einem KonkreteStrategieX-Objekt konfiguriert, verwaltet eine Referenz auf ein Objekt einer konkreten Strategieklasse und kann eine Schnittstelle für den Zugriff auf seine Daten definieren.
  - **Strategie:** Schnittstellendeklarationen für alle unterstützten Algorithmen
  - **Konkrete Strategie:** Implementiert einen Algorithmus unter Verwendung der Strategieschnittstelle



Das Strategie-Muster (Strategy Pattern)

- Konsequenzen:
  - ⇨ Strategien ersetzen Bedingungsanweisungen
  - ⇨ Bei Hinzufügung einer konkreten Strategie
    - keine Änderung im Kontext (TextVerarbeitung)
    - kleine Änderung bei der Instanziierung und Übergabe der Strategieklasse

- Beispielcode **mit** und **ohne** Strategieobjekt

```
void TextVerarbeitung::Repariere() {
  mein_formatierer->Formatiere(...);
}
```

```
void TextVerarbeitung::Repariere() {
  switch(_umbruchStrategie) {
  case EinfacheStrategie:
    FormatiereMitEinfacherStrategie(...);
    break;
  case TeXStrategie:
    FormatiereMitTeXStrategie(...);
    break;
  // ...
  }
}
```



Das Strategie-Muster - Beispielcode für die Absatzformatierung (1)

```
class TextVerarbeitung {

public:
   TextVerarbeitung(Formatierer*) { ....};
   void Repariere() {
    // Vorbereitung der Übergabeparameter
    // der Formatiere-Funktion
      mein_formatierer->Formatiere( ... );
   // Darstellung gemäß der vom Formatierer
   // berechneten und zurück gelieferten Daten
   // ...
};

private:
   Formatierer*  mein_formatierer;
   ...
};
```

Hier wird die gewünschte Strategie übergeben!

```
class Formatierer {
public:
   virtual int Formatiere(... ) = 0;
protected:
   Formatierer();
};
```

...

```
class ArrayFormatierer : public Formatierer {
public:
   ArrayFormatierer(int breite);

   virtual int Formatiere(...)
    {
     .......
     }
   // ...
};
```

...



Das Strategie-Muster - Beispielcode für die Absatzformatierung (2)

⇒ Instanziierung verschiedener Kompositions-Objekte:

```
TextVerarbeitung* schnell =
   new TextVerarbeitung (new EinfacherFormatierer);

TextVerarbeitung * schick =
   new TextVerarbeitung (new TeXFormatierer);

TextVerarbeitung* icon =
   new TextVerarbeitung (new ArrayFormatierer);
```



Das Strategie-Muster - Beispielcode für die Absatzformatierung (3)

Instanzen und Verzeigerungen für Beispiel

TextVerarbeitung* icon = new TextVerarbeitung (new ArrayFormatierer);

Aufruf:

icon->Repariere(...)

Repariere ruft auf:

mein_formatierer->Formatiere(...)



Das Strategie-Muster: Eigenschaften (I)

- Zweck:
  - Das Strategiemuster ermöglicht es, einen Algorithmus unabhängig von den ihn nutzenden Klienten zu variieren
  - es können sogar mehrere Algorithmen austauschbar gemacht werden
  - Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar
- Auch bekannt als:
  - Policy
- Motivation:
  - Es gibt unterschiedliche Algorithmen für ein spezielles Problem
  - Es ist nicht wünschenswert, alle möglichen Algorithmen in einer Klasse fest zu codieren
  - Jeder Algorithmus wird als "Strategie" in einer Klasse gekapselt und ist über eine (abstrakte) Superklasse, die eine geeignete Schnittstelle zur Verfügung stellt, anwendbar



Das Strategie-Muster: Eigenschaften (II)

- Anwendbarkeit
  - ⇨ Viele verwandte Algorithmen unterscheiden sich nur in ihrem inneren Verhalten
  - ⇨ Unterschiedliche Varianten eines Algorithmus werden benötigt, z.B. mit unterschiedlichen Vor- und Nachteilen bzgl. der Geschwindigkeit und des Speicherplatzverbrauchs
  - ⇨ Daten, die für den Algorithmus relevant sind, aber dem Aufrufer nicht bekannt sein sollen, können in einer Strategie „verborgen" werden



Diskussion Strategie-Muster

- Vorteile beim Einsatz des Strategie-Musters
  - ⇨ Strategien können leicht hinzugefügt oder entfernt werden
  - ⇨ Der Aufruf wird nicht komplizierter, bloß weil es verschiedene Strategien gibt
- Nachteile
  - ⇨ Die Implementierung ist etwas schwieriger

- Diskussion:
  - ⇨ Ein Navigationssystem bietet Ihnen die schönste Route, die schnellste Route oder auch die kürzeste Route. Wie würden Sie dies als Strategiemuster umsetzen?
  - ⇨ Sie implementieren eine Klasse, die Daten sortiert. Vorerst implementieren Sie aber nur Bubblesort und wollen dies später durch Quicksort ersetzen. Würden Sie dazu ein Strategiemuster verwenden?

Noch ein Problem aus der IT Welt: Observer-Muster (kennen Sie schon)

- Stellen Sie sich vor, Sie sollen die Systemuhr für ein modernes Betriebssystem implementieren
  - ⇨ Polling (zyklische Abfrage) soll vermieden werden
  - ⇨ Bereitstellung der neuesten Zeit und des neuesten Datums
  - ⇨ Unterstützung für eine im Voraus nicht bekannte Anzahl von Interessenten (Word, Tray-Clock, ...)
  - ⇨ Die Lösung soll möglichst übertragbar sein (z.B. für den Batteriestatus)

- Entwerfen Sie Komponenten mit Verantwortlichkeiten und Beziehungen
  - ⇨ Analysieren Sie folgende Szenarien
    1. Die Uhrzeit ändert sich und soll (ohne Polling) angezeigt werden
    2. Interessenten kommen während der Laufzeit hinzu (oder fallen weg)
    3. Analoge Aufgabe: Ermitteln des Batteriestatus des Rechners



1. Die Uhrzeit ändert sich und soll (ohne Polling) angezeigt werden

- Idee: Drehe den Spieß um und "sage Bescheid", wenn sich die Zeit ändert!
  - ⇨ Bei einer Zeitänderung ruft die Systemuhr beim UhrClient eine Methode auf
  - ⇨ UhrClient besorgt sich dann von der Systemuhr die erwünschten Daten
  - ⇨ Definiere dazu beim Client eine Methode aktualisiere()
  - ⇨ kein Polling und dennoch immer die aktuelle Uhrzeit!

2. Interessenten kommen während der Laufzeit hinzu (oder fallen weg)

realisiert durch list of * Beobachter

Systemuhr

Datum
Zeit

gibStunde()
setzeStunde()
...
meldeAn(Beobachter)
meldeAb(Beobachter)
benachrichtige()

Zeit_Beobachter *

Beobachter

{abstract} aktualisiere()

Zeit_Beobachter

UhrClient

Zeit

aktualisiere()
anzeigen()

1 Zeitgeber

- Idee: Alle Interessenten (UhrClients) "registrieren" sich bei der Systemuhr
  - ⇨ Bei einer Zeitänderung ruft die Systemuhr in ihrer Methode benachrichtige() bei allen registrierten Clients die Methode aktualisiere() auf
  - ⇨ aber das geht nur, wenn alle UhrClients zu einer Klasse gehören!
  - ⇨ Einführung einer (abstrakten) Oberklasse ("Beobachter") und Verschiebung der Assoziation



3. Analoge Aufgabe für den Batteriestatus des Rechners

- Idee: Das "Registrieren und Benachrichtigen" ist ein allgemeines Verfahren – unabhängig von den bereitgestellten Daten
  - ⇨ Herausziehen dieses Teils durch Vererbung ("Subjekt")
  - ⇨ Verschieben der Assoziation zum Benachrichtigen

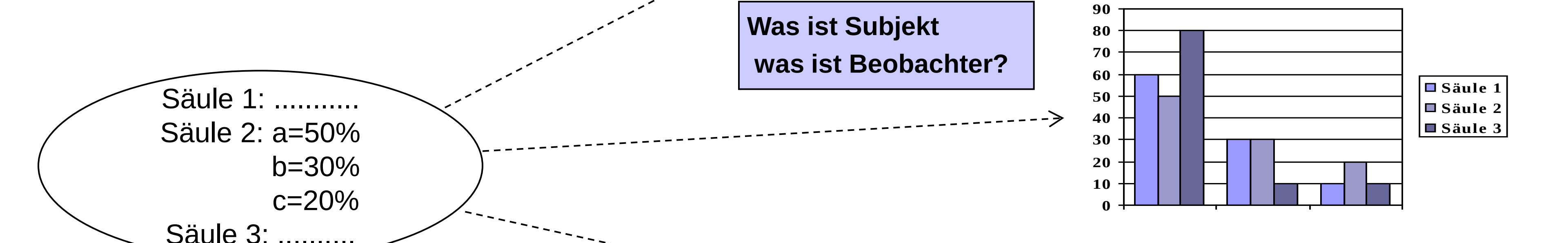
... und fertig ist das Beobachter-Muster

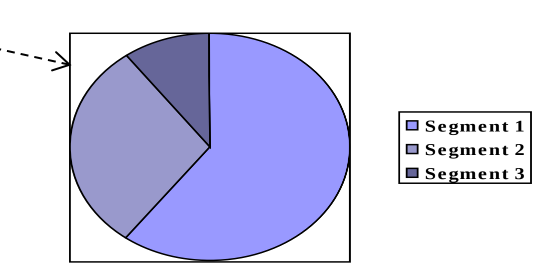


Ein anderes Problem mit der gleichen Lösung

- Mehrere Diagramme hängen von einem gemeinsamen Datenobjekt ab
- Änderungen der Daten sollen in allen Diagrammen angezeigt werden
- Die Anzahl der Diagramme ist variabel

		A	B	C
1	Säule 1	60	30	10
2	Säule 2	50	30	20
3	Säule 3	80	10	10





- Dieses Problem taucht immer wieder auf!
- Das Design-Pattern Beobachter bzw. Observer bzw. "publish-subscribe" bietet eine allgemeine Lösung dafür!



Ein anderes Problem mit der gleichen Lösung

- es gibt eine Datenbasis ("Subjekt")
- es gibt mehrere "Beobachter"
- das Subjekt benachrichtigt die Beobachter über Änderungen

		A	B	C
1	Säule 1	60	30	10
2	Säule 2	50	30	20
3	Säule 3	80	10	10

1.) Benachrichtigung über Änderung

2.) Holen der Veränderungen (Aufruf gibt Zustand zurück)



Beobachtermuster (Observer Pattern)

Bei Veränderung ⇨ benachrichtige(): für alle angemeldeten Beobachter b: {b->aktualisiere}

Nach Benachrichtigung ⇨ aktualisiere(): {beobachteter Zustand = subjekt->gibZustand}



Beobachtermuster (Observer Pattern): Aufrufverhalten

einKonkretesSubjekt: konkretesSubjekt

einKonkreterBeobachter: konkreterBeobachter

einAndererKonkreterBeobachter: konkreterBeobachter

setzeZustand()

benachrichtige()

aktualisiere()

gibZustand()

return(Zustand)

aktualisiere()

gibZustand()

return(Zustand)

für jeden angemeldeten Beobachter



Beobachtermuster (Observer Pattern): Einsatz in der Modellierung

Subjekt
meldeAn(Beobachter)
meldeAb(Beobachter)
benachrichtige()

Beobachter *

Beobachter
aktualisiere()

abstrakt

Einfach ins Modell übernehmen

1 Subjekt

KonkretesSubjekt
subjektZustand
gibZustand()
setzeZustand()

KonkreterBeobachter
beobachteterZustand
aktualisiere()

- Ein Pattern ist eine (weltweit) <u>bekannte</u> <u>Musterlösung</u>!
  - ⇨ Bitte <u>ordnen Sie die Klassen</u> auch genau <u>so an</u> (evtl. als eigenes Diagramm)
  - ⇨ <u>benennen</u> Sie die <u>Klassen Subjekt und Beobachter</u> und <u>Methoden</u> auch <u>so</u> – passen Sie nur die Parameter an



Beobachtermuster – Beispielcode Digitaluhr

Struktur:

weitere mögl. Beobachter:
z. B. Kalender, Analoguhr



Beobachtermuster – Beispielcode Digitaluhr

```
class Subjekt{
public:
 virtual ~Subjekt();
 virtual void meldeAn(Beobachter*);
 virtual void meldeAb(Beobachter*);
 virtual void benachrichtige();
protected:
 Subjekt();
private:
 Liste<Beobachter*> _beobachter;
};

void Subjekt::meldeAn(Beobachter* beobachter) {
 _beobachter->haengeAn(beobachter); //Methode von Liste
}

void Subjekt::meldeAb(Beobachter* beobachter) {
 _beobachter->entferne(beobachter);
}

void Subjekt::benachrichtige() {
 ListenIterator<Beobachter*> iter(_beobachter);
 for(iter.start(); !iter.istFertig(); iter.weiter())
 {
   iter.aktuellesElement()->aktualisiere(this);
 }                                                   };
```

```
class ZeitGeber : public Subjekt {
public:
 ZeitGeber();
 virtual int gibStunde();
 virtual int gibMinute();
 virtual int gibSekunde();
 void tick() {       // zählt um eine Sekunde hoch
                     benachrichtige() }            };
```

```
class Beobachter {
public:
 virtual ~Beobachter();
 virtual void aktualisiere(Subjekt*
          geaendSubjekt)=0; // pure virtual
protected:
 Beobachter()                                   };
```

```
class DigitalUhr: public Widget,
                  public Beobachter{
public:
 DigitalUhr(ZeitGeber* subjekt){
        _subjekt=subjekt;
        _subjekt->meldeAn(this)};
 virtual ~DigitalUhr(){...meldeAb(this)};
 virtual void aktualisiere(Subjekt* geaendSubjekt){
                ... zeichne()};
 virtual zeichne() {
        int stunde=_subjekt->gibStunde; ... }
private:
ZeitGeber* _subjekt;                            };
```

Diskussion Beobachtermuster

- Vor- und Nachteile beim Einsatz des Beobachter-Musters
  - ☺ Beobachter können leicht hinzugefügt oder entfernt werden
  - ☺ Es wird nur aktualisiert, wenn sich etwas geändert hat
  - ☹ Wenn es nur wenige und feste Beobachter sind, ist es etwas komplizierter als eine direkte Implementierung mit Abhängigkeiten
- Anwendungen:
  - ⇨ E-Mail Verteiler für SW-Updates (Benachrichtigung automatisch, Aktualisierung manuell)
  - ⇨ Anzeige der Uhr in vielen Clients auf einem PC
  - ⇨ im Praktikum: Dosierer, Mischbehälter und Display beobachten die Waage
- Diskussion:
  - ⇨ Würden Sie bei einer (traditionellen) Kuckucksuhr den Kuckuck als Beobachter der Uhr implementieren?
  - ⇨ Wie würden Sie in einem Autoradio implementieren, dass CD, Kassette, etc. bei einer Verkehrsfunkdurchsage anhalten



Das Kompositum-Muster (Composite Pattern)

Zweck:

- Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren.
- Ermöglicht es Klienten, sowohl
  - ⇨ einzelne Objekte (Blätter eines Baumes) als auch
  - ⇨ Kompositionen von Objekten (Nicht-Blatt-Knoten)

  einheitlich zu behandeln (z. B. zeichne()).

Motivation:

- Aufbau komplexer Diagramme aus einfachen Komponenten, z. B. grafische Anwendungen.
- Der Benutzer kann aus „elementaren" Komponenten (Blätter eines Baumes) größere Komponenten zusammenfügen und aus diesen wiederum größere Komponenten etc.
- ***Idee*:** Ein Client sollte bei der Verwendung der Komponenten nicht differenzieren müssen, ob es sich um eine elementare oder eine zusammengesetzte Komponente handelt.



Das Kompositum-Muster

eine rekursive Struktur aus Grafikobjekten

Zeichne(): für alle g in grafiken: g->Zeichne()

FügeHinzu(Graphik g): Füge g in Liste der Grafikobjeke ein (Liste, die Teil-von-Beziehung realisiert)

Anwendbarkeit:

- Repräsentation von Teil-Ganzes-Hierarchien
- Klient soll in der Lage sein, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren
- d. h. Klient soll alle Objekte im Kompositum einheitlich behandeln können.



Das Kompositum-Muster

Struktur:

rekursive Struktur aus Komponenten

Teilnehmer:

- Komponente: Schnittstellendeklarationen / -Implementierungen von Defaultverhalten.
- Blatt: Besitzt keine Kindobjekte,
  definiert Verhalten für die primitiven Objekte in der Komposition.
- Kompositum: Definiert Verhalten für Objekte, die Kindobjekte haben,
  speichert Kindobjektkomponenten,
  implementiert entsprechende Operationen der Schnittstelle von Komponenten.
- Klient: Manipuliert die Objekte in der Komposition durch Schnittstelle-Operationen von Komponenten.



Das Kompositum-Muster

Interaktionen:

- Klienten verwenden Schnittstellen der Klasse Komponente, um mit Objekten der Komponentenstruktur zu interagieren.
- Wenn Empfänger = Blatt
  ⇒ Direkte Abhandlung der Anfrage.
- Wenn Empfänger = Kompositum
  ⇒ Weiterleitung der Anfrage an Kindobjekte



Das Kompositum-Muster - Beispiel-Klassendiagramm

Geräte (=Komponenten) als Teil-Ganzes-Hierarchien bzw. Enthaltensein-Hierarchien.

Das Kompositum-Muster - Beispielcode (1)

```
class Geraet {
public:
   virtual ~Geraet();
   const char* Name()
         {return _name;}
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis();
   virtual FuegeHinzu(Geraet*);
   virtual Entferne(Geraet*);
   virtual Iterator<Geraet*>*
           ErzeugeIterator();
protected:
   Geraet(const char*);
private:
   const char* _name;
   Betrag Einzelteil_Preis; ...
};
```

```
class ZusammengesetztesGeraet:
      public Geraet {
public:
   virtual ~ZusammengesetztesGeraet();
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis();
   virtual FuegeHinzu(Geraet*);
   virtual Entferne(Geraet*);
   virtual Iterator<Geraet*>*
           ErzeugeIterator();
protected:
   ZusammengesetztesGeraet(const char*);
private:
   Liste<Geraet*> _teile;
};
```

nächste Seite

```
class FloppyDisk:public Geraet {
public:
   FloppyDisk(const char*);
   virtual ~FloppyDisk();
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis();
};
```

```
class Gehaeuse:
   public ZusammengesetztesGeraet {
public:
   Gehaeuse(const char*);
   virtual ~Gehaeuse();
   //virtual Watt Leistung();
   //virtual Betrag GesamtPreis();
   //virtual Betrag DiscountPreis();};
```

class Rahmen

Das Kompositum-Muster - Beispielcode (2)

```
class ZusammengesetztesGeraet:
      public Geraet {

public:
  ...
  virtual Betrag GesamtPreis();
  ...
};
```

```
Betrag ZusammengesetztesGeraet::GesamtPreis() {
  Iterator<Geraet*>* iter = ErzeugeIterator();
  Betrag gesamt = 0;

  for (iter->Start(); !iter->IstFertig(); iter->Weiter())
     gesamt += iter->AktuellesElement()->GesamtPreis();
  delete iter;
  return gesamt + Einzelteil_Preis;
}
```

Instanziierung eines Baums:

```
...
Gehaeuse *gehaeuse = new Gehaeuse("PC Gehaeuse");

Rahmen *rahmen   = new Rahmen("PC Rahmen");
gehaeuse->FuegeHinzu(rahmen);

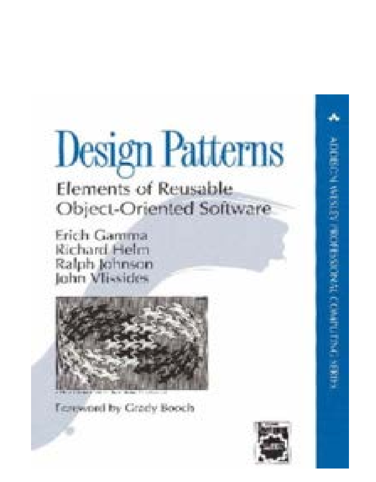
Bus* bus = new Bus("MCA Bus");
bus->FuegeHinzu(new Karte("16Mbs Token Ring"));

rahmen->FuegeHinzu(bus);
rahmen->FuegeHinzu(new FloppyDisk("3.5in Floppy"));

cout << "Der Gesamtpreis betraegt“ << gehaeuse->GesamtPreis()
   << endl;
...
```

Literatur zu Entwurfsmustern





Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
"Design Patterns – Elements of Reusable Object-Oriented Software",
(Addison-Wesley, 1994)
deutsch: Entwurfsmuster 1996
-Beispiele des Skripts stammen aus diesem Buch



Eric. Freeman & Elisabeth Freeman et.al.:
Entwurfsmuster von Kopf bis Fuß", O'REILLY, 2006



Fazit zu Entwurfsmustern

- Entwurfsmuster
  - ⇨ bieten bewährte Lösungen für immer wiederkehrende Probleme im Design
  - ⇨ sind als Lösung durch Literatur dokumentiert und international bekannt
  - ⇨ machen einen Entwurf änderbar und flexibel
  - ⇨ helfen, existierende SW-Entwürfe zu analysieren und zu reorganisieren
- Aber
  - ⇨ Entwurfsmuster machen das Design komplizierter und abstrakter
  - ⇨ es muss nicht jedes Problem mit einem Entwurfsmuster gelöst werden
  - ⇨ Entwurfsmuster sollten nur dann eingebaut werden, wenn die Flexibilität auch tatsächlich benötigt wird

Hat man ein Problem einmal selbst gelöst, weiß man das entsprechende Pattern zu schätzen - vorher versteht man leider oft nicht, warum die Lösung so gut ist !

Arten von Mustern

- Architekturmuster (Architekturstile / Architectural Patterns)
  - ⇨ dokumentieren Entwurfserfahrungen, die sich auf globale Aspekte des Gesamtsystems beziehen
  - ⇨ siehe Kapitel Architektur, z.B. 4-Schichten-SW-Architektur
- Entwurfsmuster (Design Patterns)
  - ⇨ machen lokale, sich auf wenige Klassen beziehende Entwurfserfahrungen der Wiederverwendung zugänglich
  - ⇨ nur abstrakt (UML-Modelle)
  - ⇨ oft keine Implementierung
  - ⇨ Sie kennen Beobachtermuster, Strategiemuster, Kompositum-Muster
- Software Development Antipatterns
  - ⇨ demonstrieren häufig gemachte Entwurfsfehler (nächstes Kapitel)
- Rahmenwerke (Frameworks)
  - ⇨ erlauben Wiederverwendung von ganzen Entwürfen
  - ⇨ mit teilweiser Implementierung
  - ⇨ umfassen wesentliche Aspekte einer Gruppe ähnlicher Anwendungssysteme
  - ⇨ siehe Programmieren. Sie kennen: Visual C++, später: C++Unit

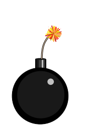


Hochschule Darmstadt
Fachbereich Informatik

Objektorientierte Analyse und Design

6.4 Anti Patterns



Software Development Anti Patterns 



- *While patterns help you to identify and implement procedures, designs, and codes that work, Anti Patterns do the exact opposite;* (Brown, Malveau, McCormick, Mitp-Verlag 2004)

  *www.**antipatterns.com/catalog**.htm*



The Blob - ein Beispiel

Catalog
- Topic
- Inventory
- ...

Person
- Name
- User_ID
- Items_Out
- Fines
- ...

Library_Main_Control
- Current_Catalog
- Current_Item
- User_ID
- Fine_Amount
- Etc.
- ...

Methods:
- Do_Inventory
- Check_Out_Item(Item)
- Check_In_Item(Item)
- Add_Item(Item)
- Delete_Item(Item)
- Print_Catalog
- Sort_Catalog
- Search_Catalog(Params)
- Edit_Item
- Find_Item
- Print
- Open_Library
- List_Catalogs
- Issue_Library_Card
- Archive_Catalogs
- Calculate_Late_Fine
- ...

Item
- Titel
- ISBN
- Author
- Publisher
- Cost
- Date_In
- Qty
- etc.
- ...

Associations: Library_Main_Control 1 → * Catalog; Library_Main_Control 1 → * Person; Library_Main_Control 1 → * Item

www.antipatterns.com/catalog.htm

- Funktionserfüllung ?
- Zuverlässigkeit ?
- Wartbarkeit ☹
- Erweiterbarkeit ☹

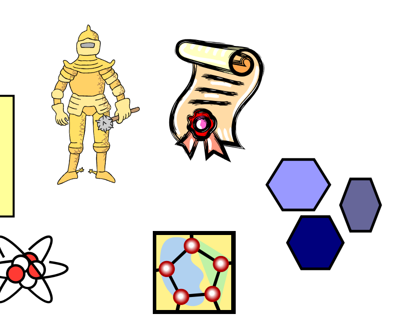
Frage:
Was ist schlecht,
Was kann man besser machen?



The Blob - ein Beispiel 

- Typisch für den Blob ist eine Klasse, welche die gesamte Verarbeitung als Monopol verwaltet, während andere Klassen primär die Daten kapseln
  - ⇨ fast die gesamte Verantwortung liegt bei einer einzelnen Klasse
    => unübersichtlich
- Im Klassendiagramm drückt sich dies dadurch aus, dass eine komplexe „Controller Klasse“ umgeben ist von zahlreichen einfachen Datenklassen
  - ⇨ "fette Spinne im Netz"
- Im Grunde genommen entspricht der Blob einem prozeduralen Entwurf, der mit Hilfe einer objekt-orientierten Sprache implementiert wird

> ⇨ Was bedeutet das für die Entwurfsprinzipien ?





The Blob - ein Software Development Anti Pattern 

- Symptome
  - **Einzelne Klasse mit einer großen Anzahl von Attributen, Operationen** oder beidem. Mehr als 50 Attribute und Operationen => Blob!?
  - Fehlendes objekt-orientiertes Design. Die einzige **Controller-Klasse kapselt die gesamte Funktionalität** des Programms, ähnlich einer prozeduralen Main-Funktion
  - Eine Klasse enthält eine Ansammlung der unterschiedlichsten **Attribute und Operationen, die keine Beziehungen untereinander haben** (Trennung von Zuständigkeiten!)
  - Eine einzige Controller-Klasse mit wenigen **assoziierten, einfachen Daten-Objekt-Klassen**
- Konsequenzen
  - Der Blob **begrenzt die Möglichkeiten,** das **System zu modifizieren** ohne die Funktionalität anderer, gekapselter Objekte zu beeinflussen (Geheimnisprinzip)
  - Typischerweise ist der Blob **zu komplex für Wiederverwendbarkeit und Tests**. Es ist nahezu ausgeschlossen, Untermengen der Funktionalität eines Blobs wieder zu verwenden
  - Es kann aufwändig sein, eine Blob-Klasse in den **Speicher zu laden**, da sie exzessiv Ressourcen verbraucht, selbst für einfache Operationen.



The Blob - Lösungsansätze (I)

- Ziel einer Neustrukturierung ist es,
  - ⇨ Verhalten aus dem Blob herauszunehmen und gekapselten Datenobjekten zuzuordnen.
    - Im Beispiel kapselt die Library-Klasse die Summe aller System-Funktionalitäten.
  - ⇨ Identifiziere oder kategorisiere zusammengehörige Attribute und Operationen
    - Die Zusammengehörigkeit sollte sich auf einen gemeinsamen Fokus, ein Verhalten oder eine Funktionalität im Gesamtsystem beziehen.
    - Wir sammeln Operationen, die Bezug haben zum Katalog-Management, wie Sort_Catalog und Search_Catalog.
    - Ebenso identifizieren wir alle Operationen und Attribute die Bezug haben zu einem einzelnen Ausleiheobjekt, wie z.B. Print_Item, Delete_Item etc.
  - ⇨ Finde eine geeignete Oberstruktur für diese neuen Gruppierungen („natural homes“) um sie dort einzubinden.
    - Im Beispiel sind dies die bereits vorhandenen Klassen Catalog und Item.



The Blob - Lösungsansätze (II)

- Löse alle redundanten oder indirekten Beziehungen und ersetze sie durch direkte Beziehungen. Im Beispiel betrifft dies die Beziehung zwischen Item und Library:
  - Jedes Item steht zunächst in unmittelbarer Beziehung zu einem Katalog und nur über diesen - also indirekt - auch in Beziehung zur gesamten Bücherei.
  - Wir verzichten auf die Beziehung zwischen Item und Library und definieren die direkte Beziehung zwischen Catalog und Item.
- Prüfe, ob assoziierte Klassen ggf. als Klassen erkannt werden können, die von einer gemeinsamen Basisklasse abgeleitet sind
- Ersetze, füge hinzu: transiente*) Beziehungen und Beziehungsklassen
  - durch Klassen, die die entsprechenden Attribute und Operationen kapseln.
  - Im Beispiel entstehen bei diesem Schritt Klassen wie Checked_out_Item.

*) transient: vorübergehend existierend



The Blob - die überarbeitete Lösung

⇨ Verbesserte Wartung, Modifizierbarkeit, Wiederverwendbarkeit...



Fazit zu Anti Patterns

- Der Blob
  - ist ein "beliebtes" Entwurfsprinzip bei Anfängern
  - einfache Kommunikation, einfache Zuständigkeit, einfache Realisierung...
  - erschwert aber Wartung, Modifizierbarkeit, Wiederverwendbarkeit etc.
  - kann leicht vermieden werden, wenn man das Anti Pattern kennt
- Regeln zum Umgang mit Anti Patterns (nach Hays W. McCormick)
  - Anti Patterns veranschaulichen häufige Fehler und bieten bessere Ansätze
  - Anti Patterns sind normale Lösungen, die den Stand der Entwicklung widerspiegeln
    - Manche Anti Patterns muss man tolerieren

Kontrollfragen Design

- Was ist die Aufgabe des Designs?
- Welche Grundprinzipien für den Entwurf kennen Sie?
- Erkennen Sie in Ihrem Praktikumsentwurf von SWE Teile des Blob?
- Warum sollte man Design-Patterns verwenden?
- Wozu sind Anti Patterns gut?
- Woran erkennen Sie ein Design-Muster bei Ihrer Arbeit?
- Wann setzen Sie das Strategiemuster ein? Erklären Sie die Funktionsweise.
- Warum kann man Strategien nicht einfach als Vererbung umsetzen?
- Woher wissen Sie welche Klassen und Operationen ihr Entwurf enthalten sollte?

Können Sie jetzt einen guten Entwurf machen ?

Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

7. Implementierung



Einordnung im V-Modell



Lernziel Implementierung

- Sie sollen in diesen Kapitel verstehen,
  - ⇨ wo der Unterschied zwischen Implementierung in "Programmieren I & II" und Implementierung in "OOAD & SWE" liegt
  - ⇨ was Programmierrichtlinien festlegen
  - ⇨ warum Programmierrichtlinien wichtig sind
  - ⇨ warum Programmierrichtlinien unbeliebt sind

Anschließend wundern Sie sich nicht mehr, warum es in einem großen Projekt so viele Regeln gibt!



Implementierung

- In Programmieren I und II haben Sie gelernt
  - Konzeption von Datenstrukturen und Algorithmen
  - Strukturierung von Programmen
  - Dokumentation der Lösung und der Implementierung
  - Umsetzung der Konzepte in C++
  - grobes Testen der entwickelten Programme
  - ⇨ allerdings für kleine Projekte in kleinen Teams ("Programmieren im Kleinen")
- Im Software Engineering geht es um "Programmieren im Großen"
  - ⇨ die erlernten Implementierungstechniken bleiben gültig
  - ⇨ aber die Randbedingungen ändern sich!
  - ⇨ Wartbarkeit und Beherrschung der Komplexität dominieren die Entwicklung!



Damit ein großes Projekt entwickelt werden kann, müssen gemeinsame Spielregeln vereinbart werden!



Bedeutung von Programmierrichtlinien

- Ziele
  - ⇨ Zusammenarbeit in einem Projekt erleichtern
  - ⇨ Integration und Test erleichtern
  - ⇨ Langjährige Wartung ermöglichen
  - ⇨ Zuverlässigkeit und Robustheit erhöhen
  - ⇨ Steigerung der Effizienz der Entwicklung
- Richtlinien
  - ⇨ sind verbindlich für den Entwickler
  - ⇨ sollten nach Möglichkeit eingehalten werden
  - ⇨ werden bei Beauftragungen zur Auflage gemacht

> ⇨ die Einhaltung der Richtlinien wird überprüft
> ⇨ Abweichungen sind nur mit (guter) Begründung erlaubt



Exemplarischer Inhalt einer Programmierrichtlinie (mit Beispielen)

- Allgemeines
  - Es ist nach dem ISO C++-Standard zu programmieren
- Kommentierung
  - Jede Klasse muss am Ort ihrer Deklaration mit einem Kommentar eingeleitet werden, der Name und Zweck der Klasse, Autor, Erstellungsdatum und Version sowie eine Änderungsliste enthält
- Namenskonventionen
  - Die C++-Schlüsselwörter sind als Symbolnamen (auch in Großbuchstaben) nicht zulässig
- Deklarationen
  - Bei paarweise existierenden Operatoren wie == und != muss, falls einer überladen wird, auch der andere überladen werden
- Fehlerbehandlung
  - Der Rückgabewert von new muss vor der ersten Verwendung geprüft werden
- Sonstiges
  - Statt der Funktionen alloc() und free() sind new und delete zu verwenden
  - Bei Bedarf sind die Standard-Datentypen string, vector und set zu verwenden



Inhalt einer Programmierrichtlinie: Weitere Themen

- Festlegung
  - ⇨ der I/O-Methoden
  - ⇨ der verwendeten Zeichensätze
  - ⇨ von erlaubten Dateinamen
  - ⇨ von Variablennamen mit Datentypen (z.B. uint32MyNumber)
- Regeln
  - ⇨ zur Verwendung von Zeigern
  - ⇨ zur Verwendung von Sichtbarkeiten / Zugriffsrechten
  - ⇨ zum Umgang mit dem Preprozessor / Compiler-Warnungen
  - ⇨ zum Aufteilung des Codes auf Dateien (z. B. eine h-Datei enthält eine Klasse)
  - ⇨ zur Art der Fehlerbehandlung
  - ⇨ zur Initialisierung von Variablen
  - ⇨ zu Layout, Formatierung des Codes
  - ⇨ und vieles mehr – je nach Projekt und Bedarf!

Vorteile einer Programmierrichtlinie

- Wartbarkeit
  - ⇨ erhöht die Verständlichkeit des Codes für Teammitglieder und Neulinge
  - ⇨ reduziert Bedarf an speziellem Fachwissen
- Integrierbarkeit
  - ⇨ Vermeidung von spät auffallenden Inkompatibilitäten von Bibliotheken
  - ⇨ Einheitliche Aufteilung des Codes auf Quellcode-Dateien
- Zuverlässigkeit
  - ⇨ klare Regeln zum Umgang mit fehlerträchtigen Konstrukten (z.B. Pointern)
- Robustheit
  - ⇨ Initialisierungsregeln, Zuweisungsregeln
- Testbarkeit
  - ⇨ Einheitliche Fehlerbehandlung
- Effizienz der Entwicklung
  - ⇨ Bietet eine Kommunikationsbasis
  - ⇨ Empfehlung von anerkannten / bewährten Elementen
  - ⇨ Vermeidung von bekannten fehleranfälligen Sprachkonstrukten

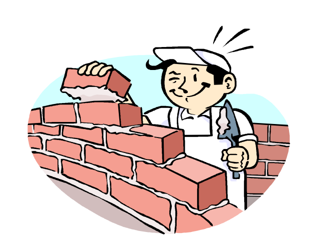


Nachteile einer Programmierrichtlinie

- Echte Nachteile:
  - ⇨ Einschränkung der Individualität des Entwicklers
  - ⇨ Gewöhnung an die vorgegebene Lösungen erforderlich



- beliebte Ausreden zur Umgehung:
  - ⇨ "Schönes Programmieren ruiniert die Performanz"
    → der Unterschied wird von heutigen Compilern weg-optimiert!
  - ⇨ "Kommentare in halbfertigem Code sind sinnlos"
    → grundlegende Entscheidungen wurden im Modell getroffen!
  - ⇨ "Die Anpassungen mache ich, wenn ich fertig bin"
    → das passiert dann nie



- Oft Widerstand – speziell von den "Programmier-Profis"

Die Richtlinien müssen als etwas Positives empfunden werden!
- ⇨ Erstellung zusammen mit den Programmier-Profis
- ⇨ unter Beteiligung aller Projektparteien
  (Entwickler, Tester, Integratoren, Qualitätssicherung…)

Kontrollfragen Implementierung

- Warum sind Programmierrichtlinien oft unbeliebt?
- Wie kann man den Widerstand der Entwickler reduzieren?
- Warum wird die Einhaltung von Regeln höher gewichtet als Effizienz, Stimmung und Entwicklungsgeschwindigkeit?
- Was passiert in einem großen Projekt, wenn es keine Programmierrichtlinien gibt?
- Wann treten erstmalig Probleme auf, wenn Sie eine andere String-Bibliothek verwenden, als die in den Richtlinien vorgegebene?

Könnten Sie jetzt in einem Team an einem großen Projekt mitarbeiten?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8. Test und Integration



Einordnung im V-Modell

Anforderungs-definition

Anwendungsszenarien

Testfälle

Abnahmetest

Grobentwurf (SW-Architektur)

Testfälle

Systemtest

Feinentwurf (Design)

Testfälle

Integrations-test

Modul-Imple-mentierung

Test-fälle

Modultest

Lernziel Test und Integration

- Sie sollen in diesen Kapitel verstehen
  - was ein Fehler ist
  - was Testen ist
  - wie man geschickt Testfälle auswählt
  - dass Black-Box und White-Box unterschiedliche Testmöglichkeiten bieten
  - welche Unterschiede es gibt zwischen Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung
  - was Integration ist
  - warum es im V-Modell 4 Testphasen gibt
  - was in den einzelnen Testphasen zu tun ist

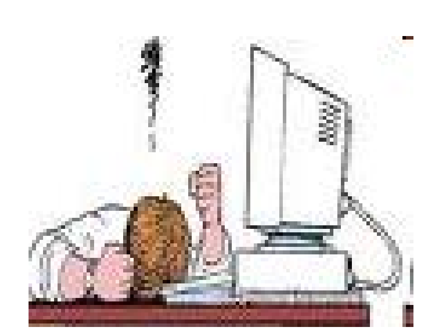
Anschließend können Sie systematisch Testfälle erstellen und anwenden!



Was ist ein Fehler?

Ein Fehler ist eine Abweichung des tatsächlichen Verhaltens vom erwarteten Verhalten





- Abstürze
- Datenverluste
- Fehlverhalten
  - ⇨ falsch implementierte Regeln / Berechnungen
  - ⇨ 1. Ursache: Programmierfehler
    - *Lösung:* Test der Programme
  - ⇨ 2. Ursache: unzureichendes Verständnis der Sachverhalte
    - *Lösung:* Nachfragen bei Unklarheiten, Missverständnissen bei Anforderungsanalyse und während des Projektes -> Spezifikation präzisieren
    - *Beispiel:* IsPrime(1) liefert false, der Test erwartet true – wer hat recht?



Was ist ein Fehler? (II)

- aber auch:
- unerfüllte Erwartungen (der Benutzer hat sich etwas anderes vorgestellt)
  - ⇨ mangelnde / unvollständige Kommunikation mit den Anwendern
    - *Lösung:* Sorgfältige Anforderungsermittlung, Prototypen
- Schlechte Performance
  - ⇨ bei Anforderungsermittlung definiert
  - ⇨ auch: technische Machbarkeit prüfen
    - *Lösung*: Performance-Prototypen, Performance-Tests
- Inkonsistente Benutzerschnittstellen
  - ⇨ stören Arbeitsfluss
  - ⇨ erzeugen Fehlbedienungen
  - ⇨ verursachen geringe Akzeptanz
  - ⇨ erzeugen hohen Schulungsbedarf
    - *Lösung:* genauer spezifizieren,
      Prototypen zum Testen der Verständlichkeit,
      Einhaltung Regeln zur Ergonomie



Wie findet man Fehler?

- Entwicklertests
  - ⇨ sollen zeigen, dass das Programm funktioniert
    - "demonstratives" Testen
  - ⇨ überprüfen die Funktion der Software auf Korrektheit im normalen Betrieb
  - ⇨ werden vom Entwickler "freiwillig" durchgeführt
  - ⇨ das kennen Sie in beschränktem Umfang aus Ihrer Entwicklertätigkeit
- Robustheitstests
  - ⇨ überprüfen die Funktion der Software auf Korrektheit am Rande bzw. außerhalb des normalen Betriebs
  - ⇨ werden oft von einem speziellen Test-Team durchgeführt
  - ⇨ versuchen systematisch Fehler aufzudecken
    - "destruktives" Testen
  - ⇨ das kennen Sie in beschränktem Umfang von der Praktikumsabnahme in Programmieren



Grundidee Testen

- Ein Test(fall) zu einem Fehler erzeugt ein vom Soll-Ergebnis abweichendes Ergebnis, wenn der Fehler vorliegt

  Frage: Wie finde ich Soll-Ergebnis?



  - ⇨ Falls Ist-Ergebnis = Soll-Ergebnis => dieser Testfall ist richtig abgelaufen und der Fehler liegt nicht vor
  - ⇨ Falls Ist-Ergebnis != Soll-Ergebnis => es liegt mindestens ein Fehler vor



Teste alle möglichen Eingabekombinationen

Frage: Was halten Sie von diesem Konzept? Ist das möglich?



Grenze des Testens

Frage: Zeit für Test eines Multiplizier-Verfahrens für 2 Zahlen a 32 Bit (0*1, 1*0, 0*2, ..) bei Testgeschwindigkeit von 10^9 Tests/Sek

- $2^{64} \approx 1{,}8 * 10^{19}$ verschiedene Eingaben
  - ⇨ Ein Testdurchlauf: 585 Jahre !
    ohne Herleitung Sollergebnisse und Vergleich mit den Sollergebnissen
    (Für einen Multiplizierer für 64-Bit-Zahlen: 10,8 Trilliarden Jahre !)





Wähle "geschickte" Stichproben !



⇨ Aber: Durch Testen kann in der Regel nicht die Fehlerfreiheit der Software nachgewiesen werden

⇨ Sondern: Testen kann nur Zuverlässigkeit verbessern



Auswahl von Testfällen

- Wie finde ich "geschickte Stichproben" (Testfälle),
  so dass mit großer Wahrscheinlichkeit viele Fehler entdeckt werden?



Für Bereiche, die sich gleich verhalten sollte ein Test durchgeführt werden.
Teste an den Stellen wo erfahrungsgemäß besonders häufig Fehler gemacht werden.



⇨ Bilde „Bereiche gleichen Verhaltens“ ("Äquivalenzklassen„) und führe Test für je einen "**Vertreter**„ durch

⇨ Fehler werden meist da gemacht, wo sich etwas ändert!

=> teste an den **Grenzen**

- von Äquivalenzklassen, von Schleifen, in Bedingungen
- In der Mitte eines Bereichs gibt es kaum Fehler

⇨ Ergänze durch "**Intuitive Testfälle**" (Erfahrung)



Kontrollfragen zum Test

- Wann ist ein Programm oder eine Funktion fehlerfrei?
- Ist es normalerweise möglich alle Testfälle laufen zu lassen?
- Was kann man tun um die Anzahl der Testfälle zu reduzieren?
- Können wir die Fehlerfreiheit durch Testen nachweisen?
- Wie erreichen wir, dass die Zuverlässigkeit nach dem Test möglichst hoch ist?
- Wo treten besonders häufig Fehler auf und wie wählt man geschickt Testfälle aus?

... und was ist eigentlich ein Testfall genau?



Inhalt eines Testfalls

- Ein Testfall ist ein Durchlauf durch das Programm.
- Erzeugt er ein vom Soll-Ergebnis abweichendes Ergebnis => es liegt ein Fehler vor
- Aber:
  - ⇨ Ein Testfall muss den Test genau spezifizieren
    => die Durchführung muss immer das gleiche Ergebnis liefern!
    Frage: z. B. Erstellung Angebot Eingabe: 3 Wasserkocher: Summe ist 30 €, beim Test in 3 Wochen: Summe ist 33 € - Warum?
    Frage: Beim Geld abheben gebe ich ein: 500 €, aber es kommt kein Geld raus! Warum? Liegt hier ein Fehler vor?
  - ⇨ Auch der Zustand – sowohl vor dem Test als auch nach dem Test - muss vollständig beschrieben werden
  - ⇨ Ein Testfall wird beschrieben durch ein Paar:
    (Eingangszustand, Eingabe) → (Sollausgabe, Sollzustand nach dem Test)



Was muss bei der Spezifikation eines Testfalls angegeben werden?

- Eingabe-/Rückgabe-Parameter einer Funktion
- Daten, die über Bildschirm ein- und ausgegeben werden
- Daten, die gelesen und geschrieben werden, z.B. von / in
- Datenbanken,
- Sensoren, externe Geräte (evtl. zu simulieren -> replizierbare Ergebnisse)
- Globale Variablen, die vom zu testenden Programm gelesen / geschrieben werden.
- Attribute von Klasseninstanzen, die vom zu testenden Programm gelesen / geschrieben werden.
- Rückgabeparameter von aufgerufenen Unterfunktionen

- Auch der Zustand des zu testenden Programms ist vollständig zu beschreiben – sowohl vor dem Test als auch nach dem Test

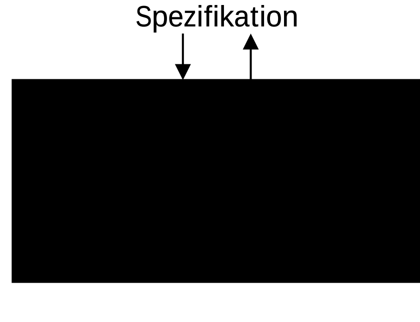
  Ein Testfall wird beschrieben durch ein Paar:
  (Eingangszustand, Eingabe) → (Sollausgabe, Sollzustand nach dem Test)



Varianten von Testobjekten

Es gibt 2 grundlegende Varianten:

- Black-Box
  - ⇨ Das zu testende Modul ist ein "schwarzer Kasten"
  - ⇨ Die innere Struktur ist für den Tester unbekannt
  - ⇨ Interne Zustände des Moduls sind nicht sichtbar
  - ⇨ Man sieht nur die Oberfläche (das "Was"), d.h. die Modulspezifikation. Nur daraus können Testfälle abgeleitet werden.
  - ⇨ z.B. externe Software-Zulieferungen als Object-File oder Abnahmetest



- White-Box
  - ⇨ Man schaut in den Kasten hinein
  - ⇨ Die innere Struktur (das "Wie") des Programms wird zur Überprüfung hinzugezogen
  - ⇨ z.B. Eigenentwicklungen

Methoden

Zweige

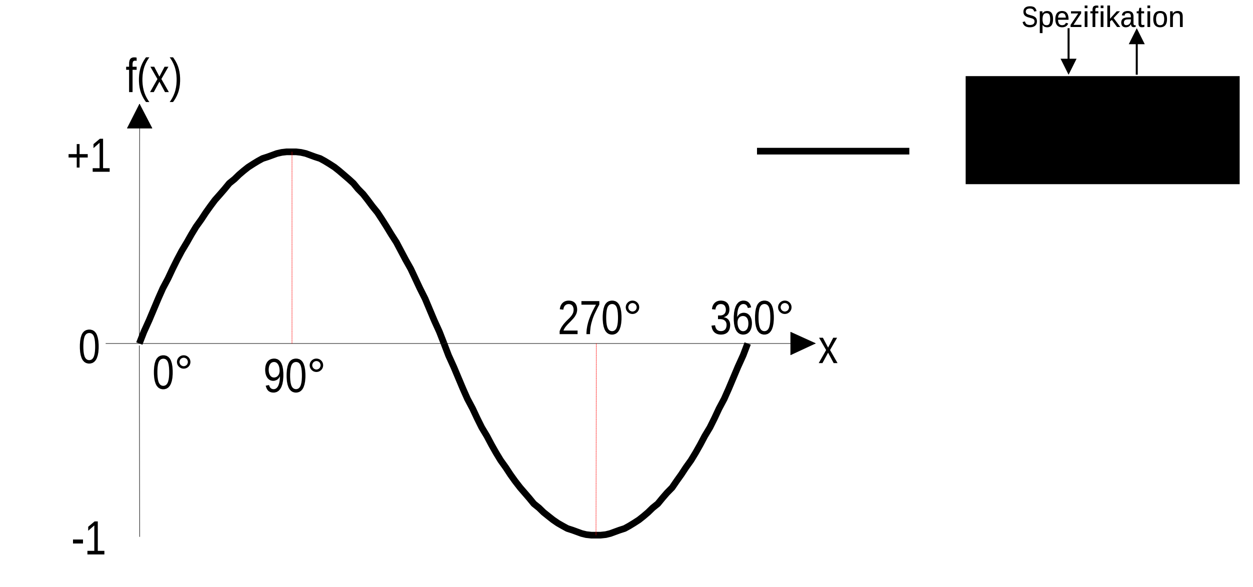
Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.1.1 Black-Box-Test



Beispiel



Über den float-Zahlen sei folgende Fkt. definiert:

f(x)=sin(x) für 0° **≤ X ≤ 360°**

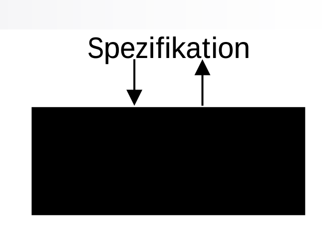
f(x)=1 für x > 360°

1. Äquivalenzklassen
   - ⇨ Es gibt vier Klassen von Eingaben: [-∞,0°[; [0°,360°] ;]360°, ∞] und Eingabe ∉ float (falls diese Eingabe möglich)

     (gültige Äquivalenzklassen sind abgeleitet aus den Fallunterscheidungen (hier jeder Zeile) der Spezifikation plus zwei ungültige Äquivalenzklassen)
   - ⇨ Teste einen Vertreter in jeder Klasse!
2. Grenzwerte
   - ⇨ An den Rändern passieren "gerne" Fehler
   - ⇨ Teste die Grenzwerte jeder Äquivalenzklasse!
3. Erfahrung, intuitive Tests
   - ⇨ Der Sinus für 90°-270° wird oft aus dem Sinus von 0°-90° berechnet, um Speicherplatz zu sparen!
   - ⇨ Teste in jedem Sinus-Segment durch einen Vertreter, ob die Berechnung stimmt



Reduzierung der Tests durch Bildung von Äquivalenzklassen

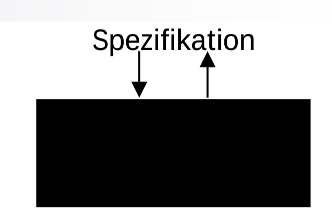


- Bilden von Äquivalenzklassen
  - ⇨ Einteilung der Menge der möglichen Werte zu einer Eingabe (z. B. Parameter) / einem Eingabezustand (z. B. das Ergebnis beeinflussendes Attribut der Klasse) in Äquivalenzklassen)
  - ⇨ Annahme: Das Testobjekt reagiert bei der Verarbeitung eines Vertreters aus einer Äquivalenzklasse "genauso" wie bei allen anderen Werten dieser Äquivalenzklasse,

    d.h. läuft das Testobjekt mit dem Repräsentanten der Äquivalenzklasse fehlerfrei, dann läuft es auch mit allen anderen Werten dieser Äquivalenzklasse fehlerfrei
  - ⇨ Beim **“Äquivalenzklassentest“** testet man das System mit jeweils einem Repräsentanten pro Äquivalenzklasse
  - ⇨ Es gibt
    - gültige Äquivalenzklassen (mit gültigen Eingabewerten) und
    - ungültige Äquivalenzklassen (mit ungültigen Eingabewerten)



Äquivalenzklassen: Beispiel Dreieck (I)



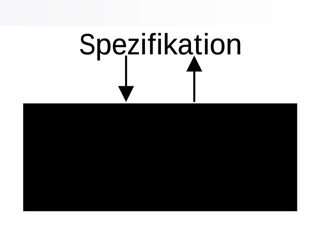
- Spezifikation der <u>Funktion "Dreieck":</u>

- DREIECKTYP dreieck (SEITE1 int, SEITE2 int, SEITE3 int);
  mit Rückgabewert enum DREIECKTYP {NORMAL, GLEICHSCHENKLIG,
  GLEICHSEITIG, KEIN_DREIECK};

- Zweck:
  - ⇨ SEITE1, SEITE2 und SEITE3 sind die <u>Seitenlängen</u> eines Dreiecks
  - ⇨ Die Seitenlängen seien <u>positive Werte</u>
  - ⇨ Die <u>Funktion stellt fest</u>, ob es sich um ein
    - einfaches (normales),
    - ein gleichschenkliges,
    - ein gleichseitiges oder
    - um gar kein Dreieck handelt
  - ⇨ Zulässige Eingabewerte seien die <u>positiven ganzen Zahlen</u>



Äquivalenzklassen: Beispiel Dreieck (II)



- ⇨ KEIN_DREIECK, wenn // 2 Seiten zusammen kürzer sind als die 3. Seite
  ((SEITE1 + SEITE2) <= (SEITE3)) or
  ((SEITE1 + SEITE3) <= (SEITE2)) or
  ((SEITE2 + SEITE3) <= (SEITE1)), sonst
- ⇨ GLEICHSEITIG, wenn // 3 Seiten gleich
  SEITE1 == SEITE2 == SEITE3, sonst
- ⇨ GLEICHSCHENKLIG, wenn // Dreieck, bei dem 2 Seiten gleich, 3. verschieden
  ((SEITE1 == SEITE2) and (SEITE1 != SEITE3)) or
  ((SEITE1 == SEITE3) and (SEITE1 != SEITE2)) or
  ((SEITE2 == SEITE3) and (SEITE2 != SEITE1)), sonst
- ⇨ EINFACH, wenn // Dreieck, bei dem alle Seiten verschieden
  (SEITE1 != SEITE2) and (SEITE1 != SEITE3) and (SEITE2 != SEITE3)

Frage: Wie sehen die Äquivalenzklassen und Testfälle aus?

Frage: können wir nicht einfach sagen: Gemäß mathematischer Definition?

- Notwendigkeit einer formalen Spezifikation:
  - ⇨ Gemäß der Mathematik beinhalten gleichschenklige Dreiecke gleichseitige Dreiecke.
  - ⇨ Die obige Spezifikation von GLEICHSCHENKLIG definiert abweichend von der mathematische Definition: Bei GLEICHSCHENKLIG müssen zwei Seiten ungleich sein, also disjunkte Aufteilung der Menge der Dreiecke!



Spezifikation

Äquivalenzklassen: Beispiel Dreieck (III)

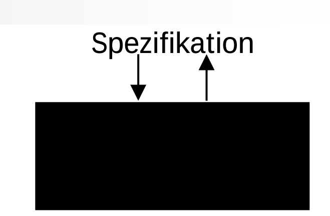


- Welche Äquivalenzklassen ergeben sich auf Grund der Spezifikation?
  - ⇨ Äquivalenzklasse 1 : Zahlen für die kein Dreieck konstruiert werden kann (eine Seitenlänge länger oder gleich der Summe der beiden anderen Seiten), Soll-Ergebnis: KEIN_DREIECK
    Repräsentant : 1,2,5
  - ⇨ Äquivalenzklasse 2 : 3 gleiche Zahlen, Soll-Ergebnis: GLEICHSEITIG
    Repräsentant : 3,3,3

  Alle folgenden Äquivalenzrelationen sind nicht KEIN_DREIECK mit folgenden zus. Bed.
  - ⇨ Äquivalenzklasse 3a : 1. und 2 Zahl gleich, 3. ungleich, Soll-Ergebnis: GLEISCHENKLIG
    Repräsentant : 3,3,4
  - ⇨ Äquivalenzklasse 3b : 1. und 3 Zahl gleich, 2. ungleich, Soll-Ergebnis: GLEISCHENKLIG
    Repräsentant : 3,4,3
  - ⇨ Äquivalenzklasse 3c : 2. und 3 Zahl gleich, 1. ungleich, Soll- Ergebnis: GLEISCHENKLIG
    Repräsentant : 4,3,3
  - ⇨ Äquivalenzklasse 4 : alle Zahlen ungleich, Soll-Ergebniss: EINFACH
    Repräsentant : 4,2,3
  - ⇨ Äquivalenzklasse 5: fehlerhafte Eingabewerte, Soll-Ergebnis: Typfehler (falls Eingabe mögl.)
    Repräsentant: -1,-15,0 (ungültige Äquivalenzklasse)



Regeln für Äquivalenzklassen

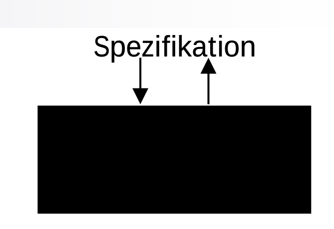


Folgende Regeln sollten beachtet werden:

- Falls Eingabetyp = Wertebereich,
  - z. B. Tage = (1 . . 31)
  - ⇨ eine gültige Äquivalenzklasse (1 ≤ Tage ≤ 31)
  - ⇨ zwei ungültige Äquivalenzklassen (Tage < 1, Tage > 31)
- Ist anzunehmen, dass Elemente einer Äquivalenzklasse unterschiedlich verarbeitet werden
  - z. B. Gleichschenkliges Dreieck
  - ⇨ evtl. Aufteilung in schmalere Äquivalenzklassen
- Falls Eingabetyp = Aufzählungstyp
  - z. B. Zahlungsart = (Scheck, Überweisung, bar)
  - ⇨ für jeden Wert eine gültige Äquivalenzklasse (mit einem Repräsentant)
  - ⇨ und eine ungültige Äquivalenzklasse (z.B. Zahlungsart = gemischt)
- Falls durch den Eingabetyp Eingabebedingungen beschrieben werden
  - z. B. erstes Zeichen muss Buchstabe sein
  - ⇨ eine gültige Äquivalenzklasse (1. Zeichen = Buchstabe)
  - ⇨ eine ungültige Äquivalenzklasse (1. Zeichen = kein Buchstabe)



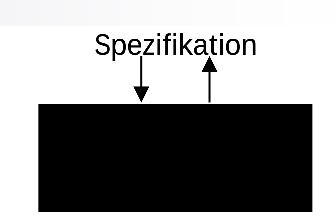
Kombination von Äquivalenzklassen



- Wie ist zu testen, falls mehrere Eingabewerte / -zustände existieren und zu jedem Eingabewert / -zustand mehrere Äquivalenzklassen existieren?
  - ⇨ Bilde das Kreuzprodukt der Äquivalenzklassen (~ N-dimensionaler Raum)
  - ⇨ Kombinationen aller kombinierbaren Repräsentanten
- z.B. bei 2 Eingabevariablen ergeben sich max. n * m Testläufe
  - ⇨ mit n, m = Anzahl der Äquivalenzklassen der Eingabevariablen

  oft weniger, da
  - ⇨ ungültige Äquivalenzklassen zu einem Eingabewert oft nur einmal getestet werden müssen, da unabhängig von anderen Eingabewerten (z. B. direkter Abbruch unabhängig von anderen Werten)
  - ⇨ Auch Eingabewerte von gültige Äquivalenzklassen können unabhängig von anderen Werten sein
  - ⇨ zwischen n * m und n + m Testläufen
- falls zu viele Testläufe
  - ⇨ Einschränkung auf Teilmenge der Eingabewertekombinationen

Ergänzung der Tests durch die Grenzwertanalyse



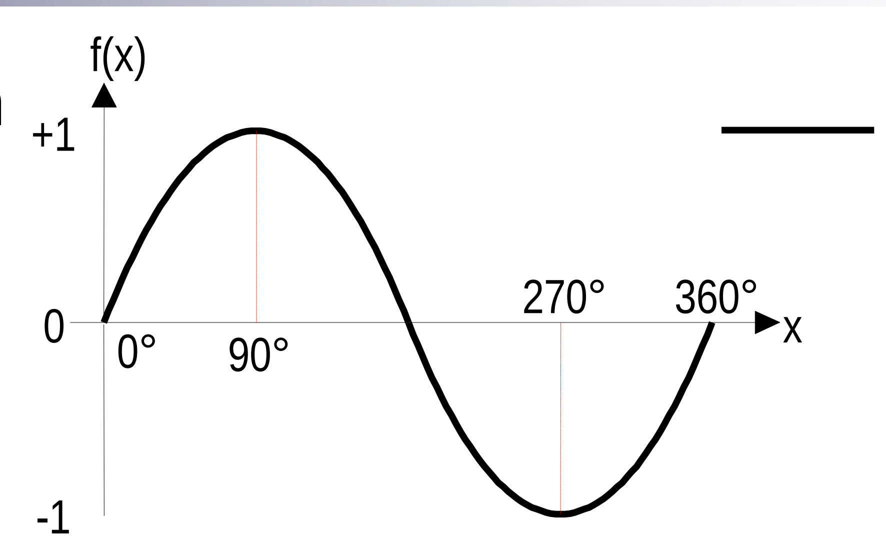
- Unterscheidung zum Äquivalenzklassentest
  - ⇨ Nicht irgendein Element aus der Äquivalenzklasse wird als Repräsentant ausgewählt, sondern ein oder mehrere Elemente, so dass jeder Rand der Äquivalenzklasse getestet wird bezüglich der
    - Eingabewerte / -zustände und
    - **Ausgabewerte / -zustände**

- Testfälle decken die Grenzwerte der Äquivalenzklassen ab
  - ⇨ Erhöhung der Quote der gefundenen Fehler
  - ⇨ Nur sinnvoll, wenn Elemente einer Äquivalenzklasse auf natürliche Weise geordnet sind, z.B.: Tage - aber nicht Zahlungsart



Spezifikation

Beispiel veränderte Sinusfunktion



$f(x)=\sin(x)$	für $0° \leq X \leq 360°$
$f(x)=1$	für $x > 360°$

1.) Äquivalenzklassen

- ⇨ Es gibt zwei gültige Klassen von Eingaben: [0°,360°] ;]360°, ∞]. Es gibt zwei Klassen von ungültigen Eingaben: [-∞,0°[; Eingaben ∉ float
- ⇨ Teste einen Vertreter in jeder Klasse. -1; 180; 370; „a“

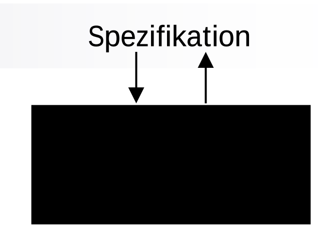
2.) Grenzwerte

- ⇨ Eingaben zu Eingabegrenzwerten: 0; 360; 360,0001; -0,0001

  (evtl. auch größte / kleinste darstellbare Zahl – abh. v. Rechner!! Ergebnis auch noch im gültigen Zahlenbereich?)
- ⇨ Eingaben zu Ausgabegrenzwerten: 90 (zu max(f(x)=1); 270 (zu min(f(x)=-1)



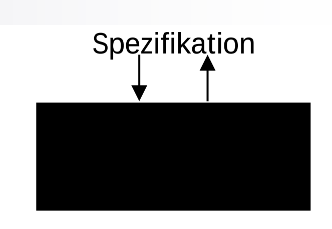
Beispiel X-Quadrat: $f(x) = x^2$ für x ist Integer



- 1.) Äquivalenzklassen: gültig: [-∞, ∞]; ungültig: Eingaben ∉ integer (falls diese Eingabe möglich)
  Representanten: 6; „a"
  2.) Grenzwerte: 0 (Ausgabegrenzwert)
- zus. zu beachten:
  - ⇨ Ist Ausgabewert im Typ des Ausgabewerts darstellbar? (< Max(Ausgabetyp)?).
  - ⇨ Weitere ungültige Äquivalenzklasse?
  - ⇨ (Ja,) Äquivalenzklassen bezüglich maximal darstellbarer Zahlenwerten für die Ausgabe (Eingabe) (Max(Ausgabetyp) ist abhängig vom Rechner!)
- => Falls Ausgabewert vom Typ Integer:
  1.) Äquivalenzklassen:
  gültig: [-int(sqr(MAX_INT)), +int(sqr(MAX_INT))] - ansonsten ist x^2 > MAX_INT
  ungültig: Eingabe ∉ integer; [-∞, -int(sqr(MAX_INT))-1] ; [int(sqr(MAX_INT))+1, +∞]
  Representanten: 6; „a"; int(-sqr(5 345 432 100)); int(sqr(5 345 432 100))
  2.) Grenzwerte?
  gültig: -int(sqr(4 294 967 296)); int(sqr(4 294 967 296)); 0 (Ausgabegrenzwert)
  ungültig: -int(sqr(4 294 967 296))-1; int(sqr(4 294 967 296))+1;
  („x ∉ integer" ist nicht geordnet => kein Grenzwert!);



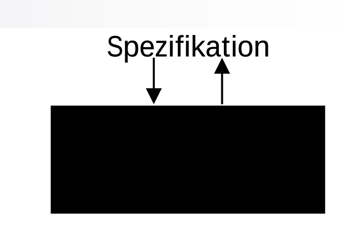
Ergänzung der Tests durch Intuitive Testfallermittlung



- Ein erfahrener Tester kennt die "beliebten" Fehler und Umsetzungsvarianten der Entwickler!
  - ⇨ der Tester macht Annahmen über die (wahrscheinliche) Umsetzung innerhalb der Black-Box und ergänzt die Testfälle entsprechend
  - ⇨ Oft findet man dabei Spezialfälle, die auch bei der Spezifikation übersehen wurden (vgl. IsPrime(1))
  - ⇨ Zum Finden von intuitive Testfällen sollten Tester und Entwickler unterschiedliche Personen sein
- Beispiele
  - ⇨ Wert 0 (bei Ein- / Ausgabewerten)
  - ⇨ Steuerzeichen in Zeichenketten
  - ⇨ Kein Eintrag oder leerer Eintrag bei Tabellenverarbeitung
  - ⇨ Einlesen von Kommazahlen in Integerfelder
  - ⇨ Gleichzeitiges oder andauerndes Drücken von Tasten
  - ⇨ …



Beispiel Fakultät



- Entwickeln Sie die Testfälle
  - ⇨ nach der Methode des Äquivalenzklassentests
  - ⇨ nach der Methode der Grenzwertanalyse
  - ⇨ Ergänzen Sie intuitive Testfälle

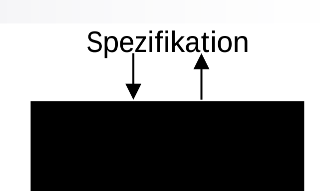
  - ⇨ FAK(n) {...} , n=integer

    Berechnung der Fakultät : n! = n * (n-1) *...3*2*1

  - ⇨ Unsere Funktion FAK(n) sei (abweichend von der math. Def.) folgendermaßen definiert:
  - ⇨ FAK(n) = -1 wenn n < 0
  - ⇨ 1 wenn n = 0 or n = 1
  - ⇨ n* FAK(n-1) sonst



Beispiel Fakultät: Lösung

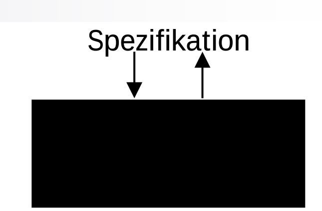


- Äquivalenzklassen
  - ⇨ gültige Klassen: [-∞,0[; [0,1] ;]1, ∞].
  - ⇨ ungültige Klassen: Eingabewert ∉ ganze Zahlen
  - ⇨ Testfälle: (-5,-1); (0,1); (5, 120); (1.5, UNDEF)
- nach der Methode der Grenzwertanalyse
  - ⇨ Testfälle: (-1,-1); (0,1); (1,1); (2,2);
- Intuitive Testfälle
  - ⇨ Die Funktion wächst so schnell, dass die Berechnung oft durch eine Wertetabelle ersetzt wird (FAK(13)>MAX_INT, FAK(21)>MAX_LONG).
  - ⇨ Eingabefehler können in jedem Wert auftreten
  - ⇨ Totaler Test möglich! => Teste jeden einzelnen Wert bis zur maximalen Größe

> Bei Ausgabewert Überschreitung des Zahlenbereichs?
> ⇒ andere Klassen-einteilung mit Berücksichtigung von MAX_INT



Beispiel Fakultät: Lösung bei Eingabe, Ausgabe = Integer



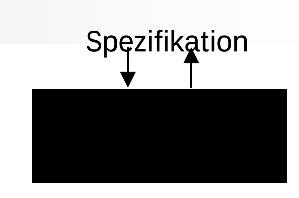
- Äquivalenzklassen
  - ⇨ gültige Klassen: [-∞,0[*) ; [0,1] ;]1, +12].
  - ⇨ ungültige Klassen: nicht ganze Zahlen; [13,+ ∞].
  - ⇨ Testfälle: (-5,-1); (0,1); (5, 120); (1.5, UNDEF); (13,UNDEF).
- nach der Methode der Grenzwertanalyse
  - ⇨ Testfälle: (-1,-1); (0,1); (1,1); (2,2); (12,479001600); (13,UNDEF).

Bei Ausgabewert Überschreitung des Zahlenbereichs?
⇒ andere Klassen-einteilung mit Berücksichtigung von MAX_INT

*) bei Eingabewert = -∞, + ∞ kann natürlich nur maximal -MAX_INT, +MAX_INT eingegeben werden -> Grenzwert?



Black-Box-Test – Beispiel Suchen in Liste



Zu testende Funktion: void suche (int Wert, int & Nr_des_Elementes) **in der Klasse Liste**

in der Klasse Liste

Eingabewert: Integer-Wert, Eingabezustand: Liste.

Ausgaben: Nr. des gefundenen Elements (int). Falls nicht gefunden: -1

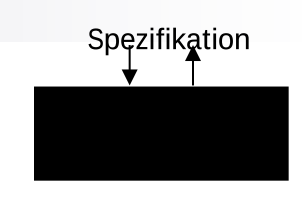
```
class Liste
{...
    void suche (int Eingabe, int &Nr_des_Elementes) ...
    void anfuege (int Element)  ...
}
```

Welche Äquivalenzklassen (abhängig von welchen Werten)?

Welche Testfälle?

Wie sieht das Testprogramm aus?

Black-Box-Test – Beispiel Suchen in Liste



Zu testende Funktion: void suche (int Wert, int &Nr_des_Elementes) **in der Klasse Liste**

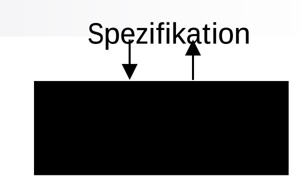
Welche Äquivalenzklassen (abhängig von welchen Werten)?
Welche Testfälle?

Eingabe:

- Listenlänge = 1 (Ein Wert in Liste) (Intuitiv) u.
  - (eingegebener) Wert in Liste (hier: 6) (Grenzwert der Ausgabe : Nr_des_Elementes=1)
  - Wert nicht in Liste (hier: 7) (Ausgabe: Nr_des_Elementes=-1)
- Listenlänge länger als 1 (hier: Einträge in Liste: 6,4,3,2) und
  - Wert = erstes Element der Folge (hier: Wert = 6) (Grenzwert der Ausgabe (Nr_des_El..))
  - Wert = mittleres Element der Folge (hier: . 2. Element, Wert = 4)) (intuitiv)
  - Wert = letztes Element der Folge (hier: Wert = 2) (Grenzwert der Ausgabe (Nr_des_El..))
  - Wert nicht in Folge (hier: Wert = 10) (Ausgabe: Nr_des_Elementes=-1)
- Listenlänge = 0 (Grenzwert des Eingabezustands (Listenlaenge)) und
  - Wert nicht in Liste (hier: 6)
- unabhängig von Listenlänge
  - Wert ein Character (hier: ‘W‘)



Black-Box-Test – Beispiel Suchen in Liste



Zu testende Funktion: void suche (int Wert, int &Nr_des_Elementes) **in der Klasse Liste**

Wie sieht das Testprogramm aus?

```
//Testprogramm zu Testfall 1
...
Liste dieListe;
dieListe.anfuege(6);
int Nr_des_Elementes;
dieListe.suche (6, Nr_des_Elementes);
if not Nr_des_Elementes ==1
   cout <<Testfall 1 mit Fehler;
```

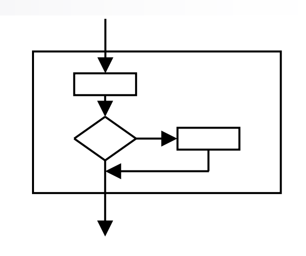
Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.1.2 White-Box-Test



White-Box-Test



- Alternative Begriffe: Glas-Box-Test, Strukturtest
  - ⇨ Man schaut in den Kasten hinein
  - ⇨ Die innere Struktur (das "Wie") des Programms wird zur Überprüfung hinzugezogen

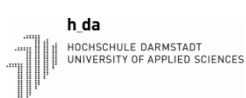
- (Zusätzliche) Testfälle werden aus der inneren Struktur abgeleitet

  Es wird – je nach Testziel - geprüft,
  - ⇨ ob alle
    - Anweisungen,
    - Zweige,
    - Pfade

    des Programms durchlaufen werden bzw.

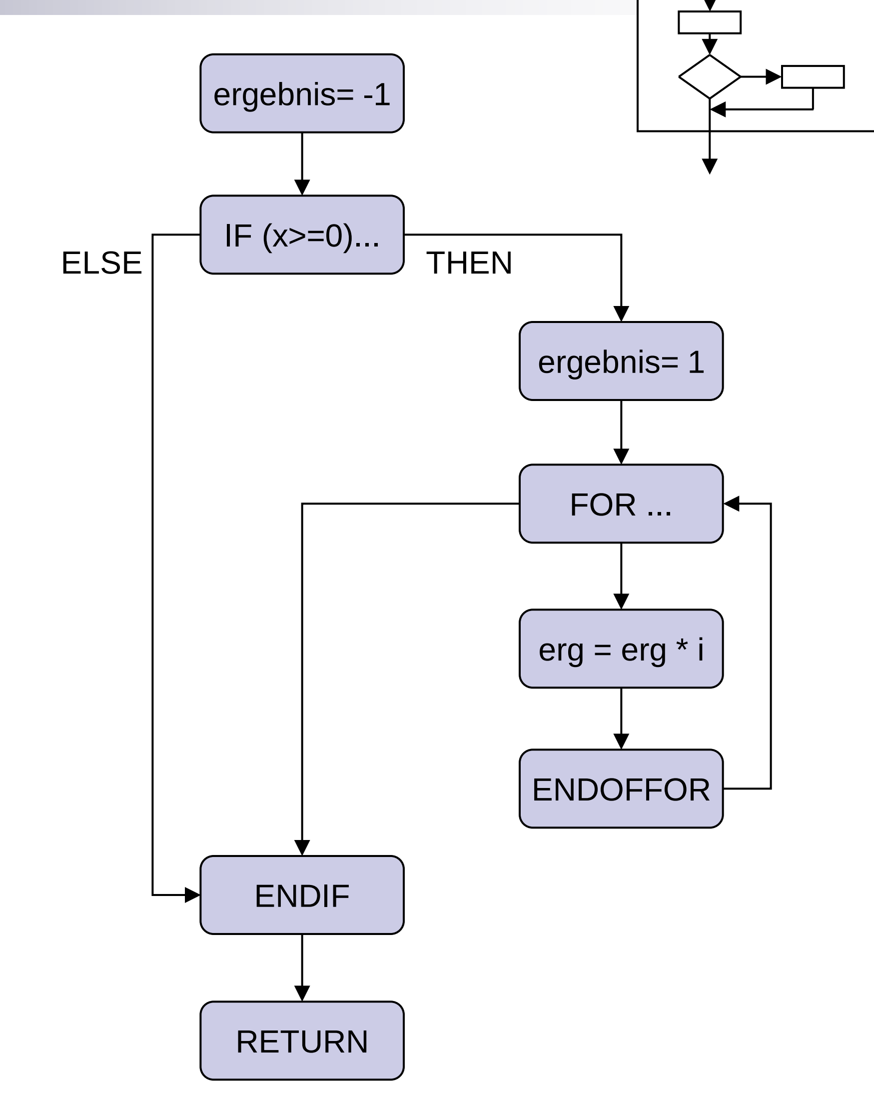
    ob alle
    - Teilausdrücke von zusammengesetzten Bedingungen mindestens einmal true und einmal false sind

Auch für eine White-Box können Blackbox-Tests angewendet werden!



Codebeispiel

```
int fak (x) {
        ergebnis = -1;
        if (x >= 0) {
            ergebnis = 1;
            for (i=2; i <= x; i++) {
                ergebnis = ergebnis * i;
            }
        }
        // else { }
        return ergebnis;
}
```



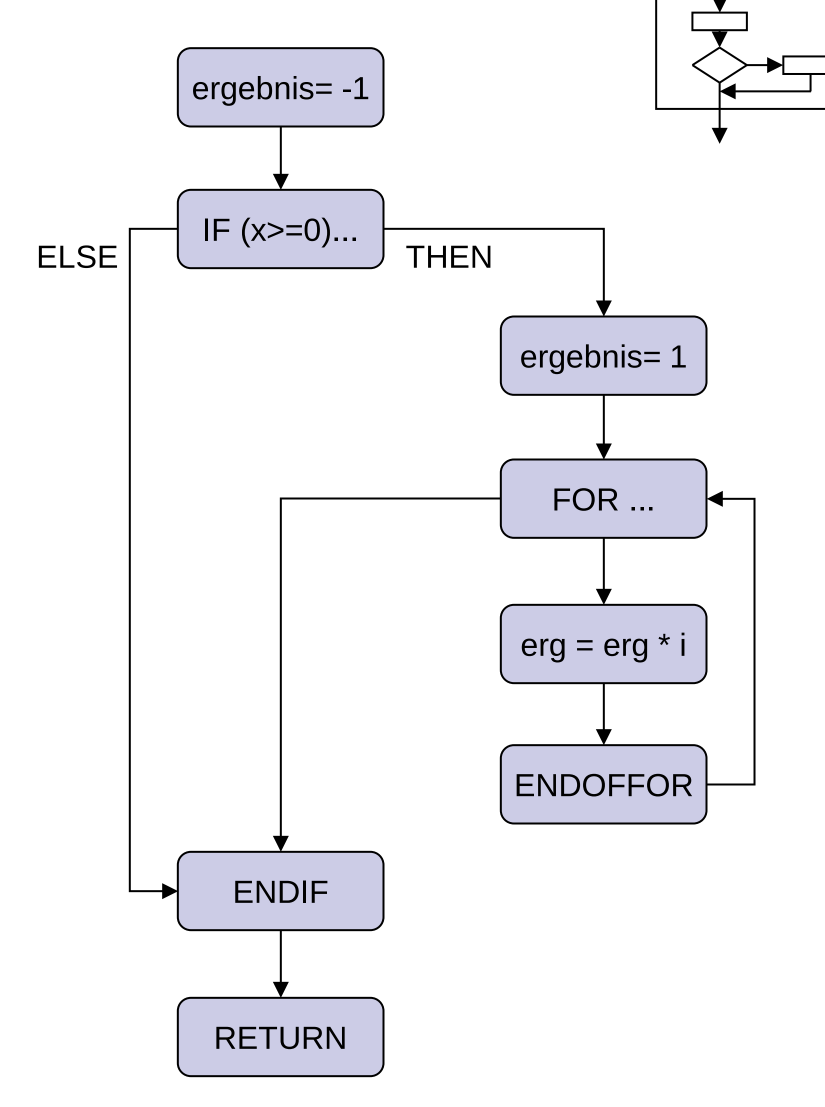
Frage: Was ist vollständige Anweisungsüberdeckung?

Kontrollflussgraph / gerichteter Programmgraph der Berechnung der Fakultät



Testziele (I)

- Vollständige Anweisungsüberdeckung
  - Jede Anweisung des zu testenden Programms wird mindestens einmal ausgeführt
  - Bedeutung:
    Wichtig für Laufzeitfehler (z.B. Speicherübergriffe, uninitialisierte Variablen etc.)



Kontrollflussgraph / gerichteter Programmgraph der Berechnung der Fakultät

Frage: Wie viele Testfälle?
Was gebe ich als Parameter ein?

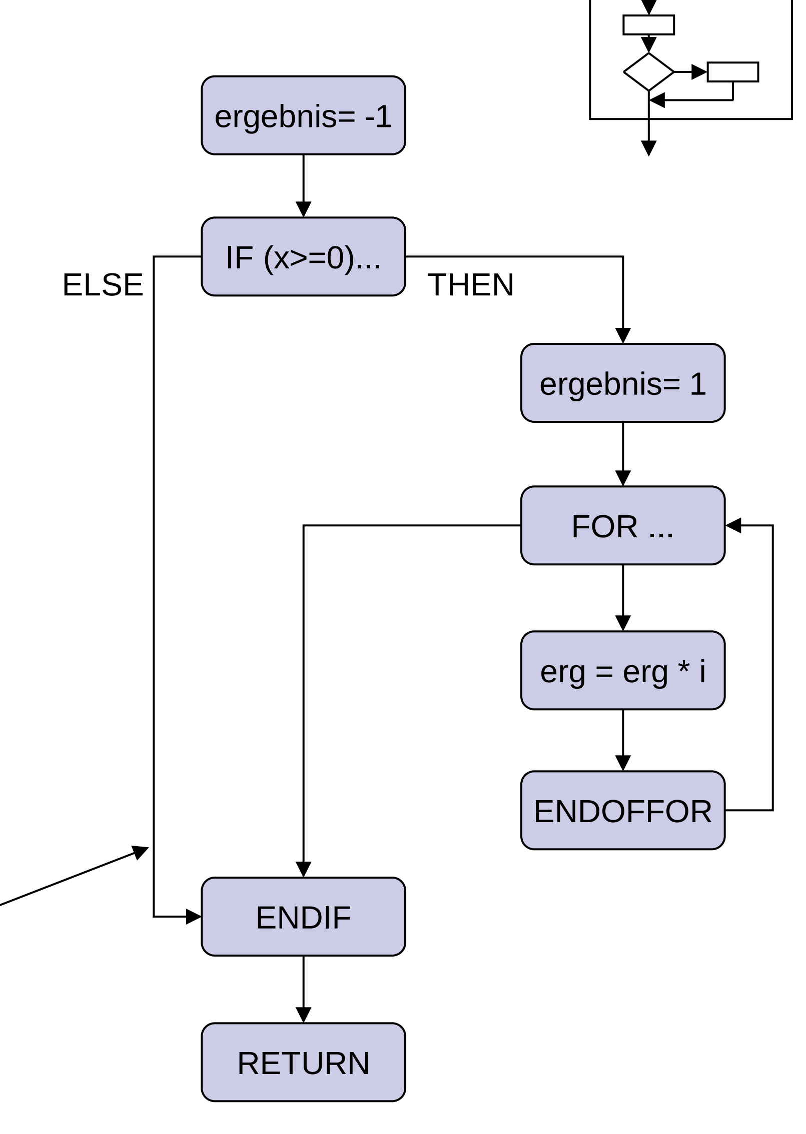
Was ist Zweigüberdeckung?



Testziele (II)

- Vollständige Zweigüberdeckung
  - ⇨ Jeder Zweig des Programms wird mindestens einmal durchlaufen
  - ⇨ Zweig = Kante (⟶) im gerichteten Programmgraphen
  - ⇨ Ein gerichteter Programmgraph, ist ein Graph, der jeden möglichen Kontrollfluss des Programms enthält (z. B. Programm-Ablauf-Plan(PAP), Aktivitätsdiagramm)
  - ⇨ Ein If hat in dieser Darstellung immer einen Then- und einen Else-Zweig (auch wenn dieser im Code nicht explizit angegeben wurde)
  - ⇨ Frage: Unterschied zu Anweisungsüberdeckung?

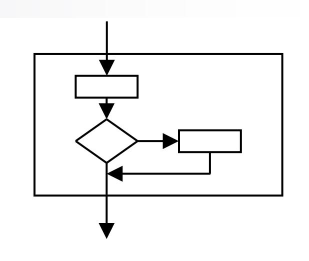
    Die Zweigüberdeckung testet zusätzlich "leere" Else-Zweige



Kontrollflussgraph / gerichteter Programmgraph der Berechnung der Fakultät

Frage: Wie viele Testfälle?
Was gebe ich als Parameter ein?
Was ist Pfadüberdeckung?

Testziele (II)



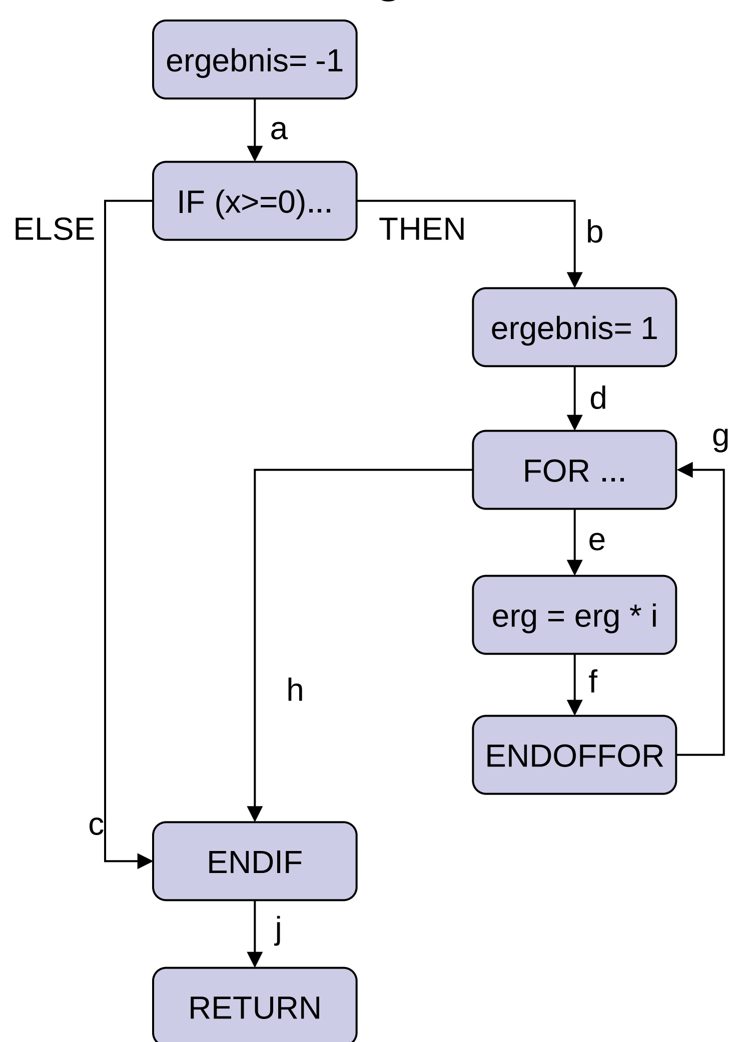
- Vollständige Pfadüberdeckung
  - ⇨ Jeder Pfad im Ablauf des Programms wird mindestens einmal durchlaufen
  - ⇨ Pfad = Weg durch den "Ausführungsbaum" von seinem **Anfang bis zu einem Ende**
    - Der Ausführungsbaum enthält alle möglichen Abläufe im Programm
    - Schleifen werden mehrfach durchlaufen.
    - Beispiel: Ein Programm mit einer Schleife, die 1 bis 3 Mal durchlaufen werden kann, muss zumindest 3 Mal getestet werden, so dass die Schleife beim 1. Testlauf 1 Mal, beim 2. Testlauf 2 Mal und beim 3. Testlauf 3 Mal durchlaufen wird.
  - ⇨ Bedeutung:
Sehr gründliche Tests, aber schnell sehr große Menge an Testfällen. (Falls eine Schleife beliebig oft durchlaufen werden darf, bräuchten wir unendlich viele Testfälle!)

Frage: Bei Funktion fak(n):

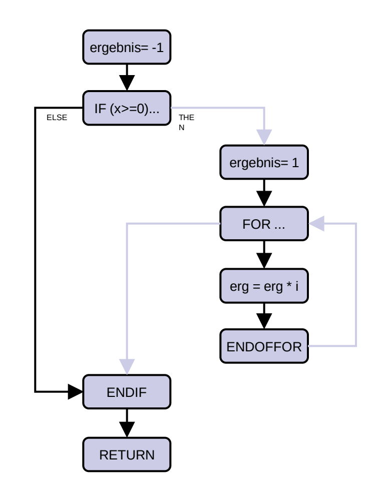
Wie viele Testfälle? Was gebe ich ein?



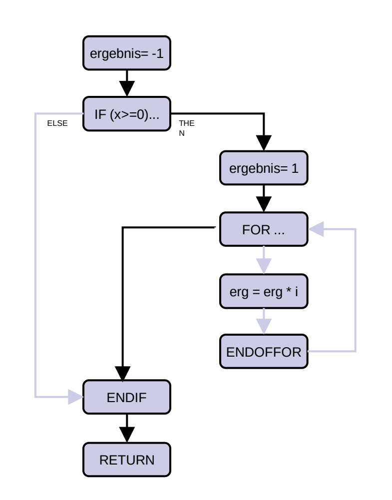
Pfadüberdeckung



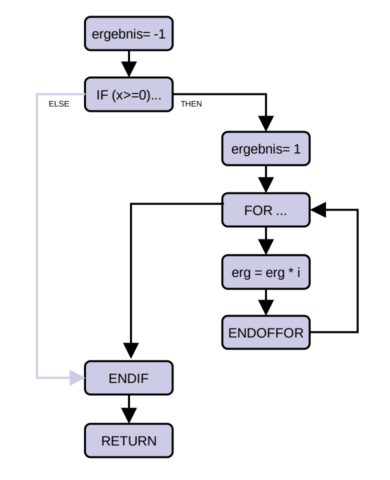
Kontrollflussgraph der Fakultäts-Berechnung



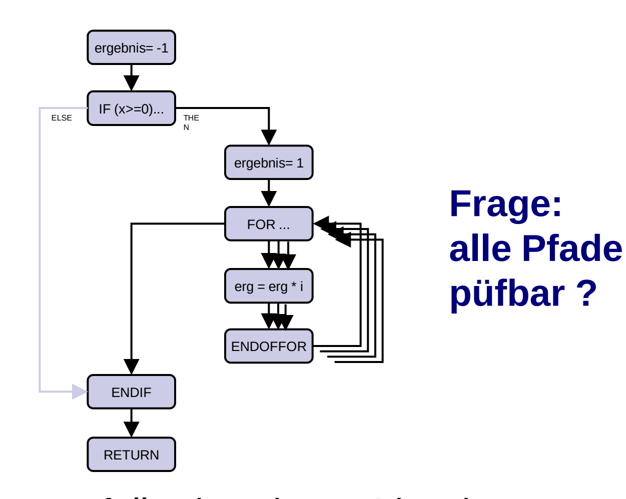
Testfall 1: x = -1 (a, c, j)



Testfall 2: x = 0 (a, b, d, h, k, j)



Testfall 3: x = 2
(a, b, d, e, f, g, h, k, j)

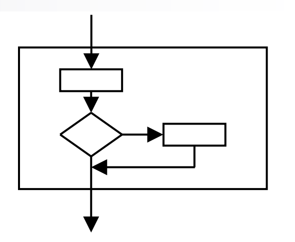


Testfall 4 (,5,...): x = 3(,4,..)
(a, b, d, [e, f, g] ^{i}, h, k, j) , i=2(,3,...)

Frage: alle Pfade püfbar ?



Pfadüberdeckung - Ausführungsbaum



ergebnis= -1
IF (x>=0)...
THE
N
ENDIF
RETURN
ergebnis= 1
FOR ...
ENDIF
RETURN
erg = erg * i
ENDOFFOR
FOR ...
ENDIF
RETURN
erg = erg * i
ENDOFFOR
FOR ...
ENDIF
RETURN
erg = erg * i
ENDOFFOR
FOR ...
ENDIF
RETURN
erg = erg * i
ENDOFFOR
FOR ...
ENDIF
RETURN
...

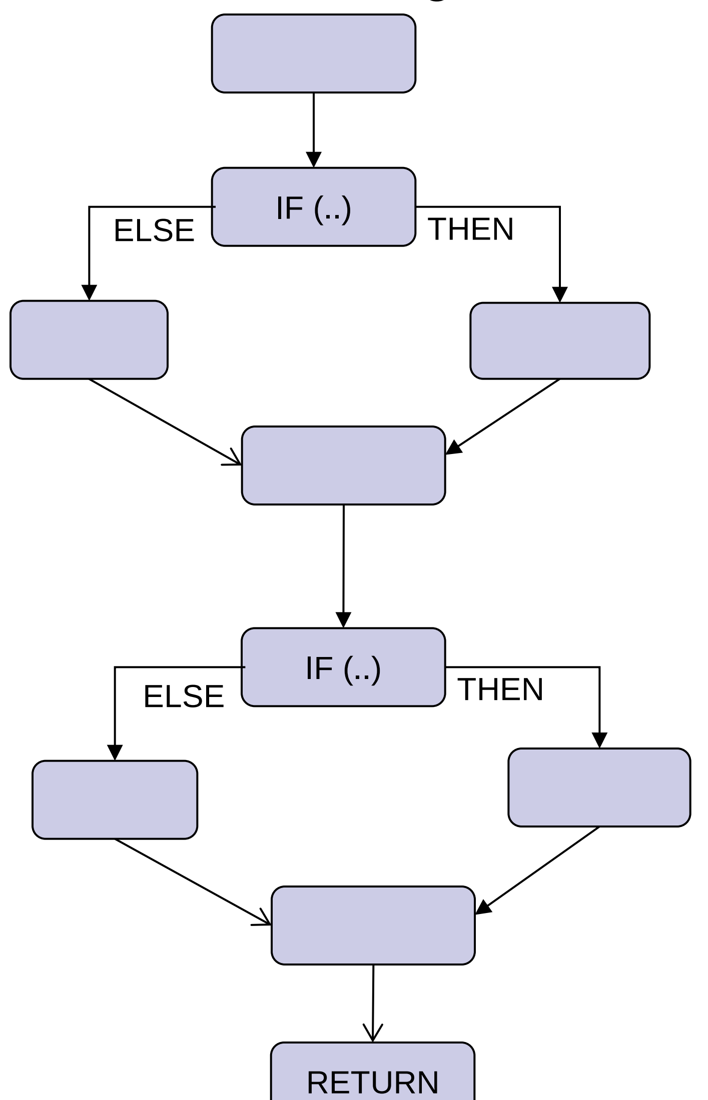
Testfall 1, .., 6, ..:

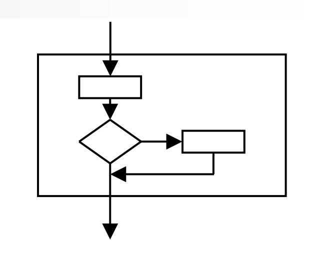
x = -1: (a, c, j)

x = 0, ..,5, .. : (a, b, d, [e, f, g] ^{i}, h, j)



Pfadüberdeckung



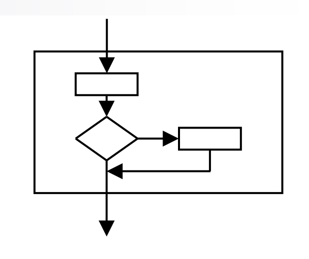


Frage: Wie oft muss dieses Programm bei vollständiger Pfadüberdeckung getestet werden?

4 Mal



Testziele (III)



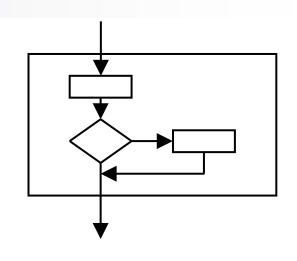
- Vollständige Bedingungsüberdeckung
  - ⇨ Jede Teilbedingung (und Gesamtbedingung) in einer Abfrage nimmt mindestens einmal den Wert true und mindestens einmal den Wert false an
  - ⇨ Beispiel: IF (Z=="A") OR (Z=="E") OR (Z=="I") → Testfälle: A, E, I, X
  - ⇨ Bedeutung:
    Fehler bei der Erstellung von Bedingungen werden erkannt

Bemerkung:

Für eine Bedingungsüberdeckung langt nicht, dass jede Gesamtbedingung mindestens einmal den Wert true und mindestens einmal den Wert false annimmt (= Zweigüberdeckung!)



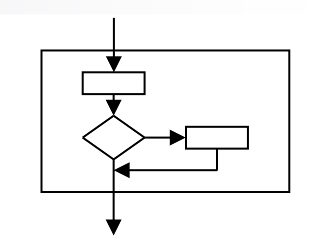
Vergleich der Testprinzipien



Ergebnisse für vollständige Abdeckung	Anweisungs-überdeckung	Zweig-überdeckung	Pfad-überdeckung	Bedingungs-überdeckung
Fehlerhafte Anweisung	☺	☺	☺	☺
"leere" Else-Anweisung	☹	☺	☺	☺
schwierige Fehler in IF-Bedingungen	☹	☹	☺	😐
Entdecken von fehlerhaften Teilbedingungen im IF	☹	😐	😐	☺
Rand-Fehler in komplexer Bedingung (x>0 statt x ≥0)	☹	☹	☹	☹
Anzahl von Testfällen	viele	viele	sehr viele	viele

☺ wird entdeckt
☹ wird nicht entdeckt
😐 wird tlw. entdeckt

Kombination von Testprinzipien



- Häufig treten in der Software Fehler auf, weil eine If-Abfrage an der Grenze zwischen den beiden Fällen unsauber formuliert ist (z.B. x>0 statt x≥0)



Kombiniere die Zweigüberdeckung mit der Grenzwertanalyse!

- d.h. teste nicht nur irgend einen Wert pro Zweig, sondern die Grenzwerte!
  - ⇨ Werte, die - falls sie etwas größer bzw. kleiner wären - den Ablauf in den anderen Zweig steuern würden („Ränder von Schleifen“)



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.1.3 Testdurchführung



Allgemeine Strategie

1) Testfälle nach Black-Box-Test suchen

2) Wenn möglich - ergänzende Testfälle nach White-Box-Test suchen

   nach Zweigüberdeckung

   evtl. zusätzliche Testfälle gemäß Bedingungsüberdeckung

   evtl. teilweise Pfadüberdeckung

3) Zusätzlich intuitive Testfälle suchen

- Testziel
  - ⇨ Ziel ist es so viele Fehler wie möglich zu finden
    - und nicht die Fehlerfreiheit nachzuweisen!
  - ⇨ psychologischer Grund: Prüfer muss motiviert werden, Fehler zu finden!



Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!"





Ablage von Testdaten

- Tests müssen reproduzierbar sein
  - ⇨ die Funktion muss erneut getestet werden können
    - z. B. nach einer Programmänderung
    - Eingabe- und Ausgabedaten müssen aufgehoben werden
- Ablage und Testautomatisierung mit Hilfe von Testwerkzeugen
  - ⇨ Bei Erstellung des Testplans
    - Ein- / Ausgabedaten und Vor- / Nachzustände werden hergeleitet.
  - ⇨ Beim Testlauf (für den Test eines Testfalls)
    - Durch eine Setup-Funktion wird die Testumgebung erstellt (z. B. Instanziierung von Klassen etc.) und die Vorzustände (z. B. Attributwerte von Klasseninstanzen, globale Variablen, Datenbankzustände etc.) werden gesetzt.
    - Das Testwerkzeug ruft die Funktion mit vorgegebenen Eingabedaten auf und
    - testet das Ergebnis auf Korrektheit. (Ergebnis muss manuell hergeleitet werden!)
    - Evtl. müssen auch Nachzustände auf Korrektheit geprüft werden.
  - ⇨ Nach Testlauf:
    - Das Testwerkzeug räumt auf (Tear-down-Funktion), d. h. durch new instanziierte Klassen müssen destruiert werden etc.



Toolunterstützung

- Test-Tools
  - ⇨ erleichtern die Erstellung von Testfällen und das Aufsetzen der Zustände für den Test (z.B. JUnit, C++Unit, ...)
  - ⇨ verwalten Testfälle und automatisieren die Test-Durchführung
  - ⇨ "markieren" nach der Durchführung einer Menge von Testfällen die durchlaufenen Anweisungen, Zweige und Bedingungen
    - man muss "nur noch" fehlende Tests ergänzen
    - erleichtert schnelle Erkennung von schwer / nicht testbaren Programmteilen
  - ⇨ Bei schlechter Überdeckung:
    - Überarbeitung des Codes zur Verbesserung der Testbarkeit
    - Entfernen von unbenutztem Code
    - Einbau von Zugriffsmöglichkeiten zu Testzwecken
    - ⇨"Design for Testability"

Ohne Tool sind systematische Tests für größere Programme kaum machbar!



Einordnung im V-Modell

Anforderungs-definition

Anwendungsszenarien

Abnahmetest

Grobentwurf (SW-Architektur)

Testfälle

Systemtest

Feinentwurf (Design)

Testfälle

Integrations-test

Modul-Imple-mentierung

Test-fälle

Modultest

Die Testspezifikation sollte jeweils an den Phasenenden erfolgen!

Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.2 Modultest



Was ist ein Modultest?

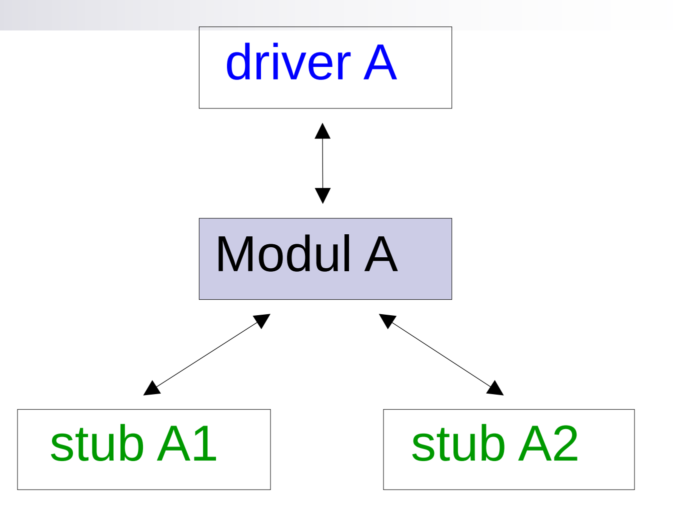
- Ein Modultest ist der Test eines einzelnen Teilstücks der Software - eines Moduls:
  - ⇨ Funktionen
    - lokales Testen von globalen Funktionen / Funktionen in Klassen
  - ⇨ Klassen
    - Testen aller Funktionen (Schnittstelle) mit Aufruf von Funktionen der gleichen Klasse.
  - ⇨ Ketten von verbundenen Klassen z.B. durch Assoziationen / gegenseitige Aufrufe
    - Testen von Funktionen mit Aufruf von Funktionen anderer Klasseninstanzen
  - ⇨ Komponenten
    - Testen aller Funktionen der Schnittstelle der Komponente.

Jede Funktion wird zuerst einzeln für sich getestet! Aber wie?



Simuliere die Umgebung des Moduls durch Test-Module.
Die aufrufende Module ersetze ich durch „Driver“,
die aufgerufene Module durch "Stubs"

Isolierter Test einzelner Funktionen



- "Drivers" sind Testmodule, die
  - ⇨ das Verhalten einer aufrufenden Funktion simulieren
    - Testumgebung erstellen
    - Eingangszustände setzen (Datenbank, Datei, Attribute von Klassen, globale Variablen ...)
    - die zu testende(n) Funktion(en) des Moduls aufrufen und mit Parametern versorgen,
    - Bildschirmeingaben definieren oder durch Umleitung des Streams automatisch einspielen
    - Ergebnisse entgegennehmen
    - Ergebnisse (Ausgabeparameter, Bildschirmausgaben) und Ausgangszustände (Datenbank, Datei, Klassenattribute, globale Variablen ...) prüfen bzw. den Benutzer bei der Prüfung unterstützen
    - System in Zustand vor dem Test zurückversetzen
- "Stubs" sind Testfunktionen, die
  - ⇨ das Antwort-Verhalten einer echten aufgerufenen Funktion (mehr oder weniger) simulieren
    - Parameter von der zu testenden Funktion entgegennehmen
    - Ergebnisparameter (die evtl. "falsche" Werte besitzen können) nach oben zurückgeben



Besonderheiten beim Modultest

- Test von Funktionen in Klassen:
  - ⇨ Instanziieren von Klasseninstanzen
  - ⇨ Eine Funktion hat evtl. auch Attribute von Klasseninstanzen als Eingabe- oder Ausgabezustände.
  - ⇨ Beim Test berücksichtigen!

- Aufrufhierarchien von mehreren Funktionen
  - ⇨ Es werden einzeln getestete Funktionen zu Aufrufhierarchien (auch über Klassengrenzen hinweg) zusammengeführt
    - innerhalb einer Klasseninstanz:

      Teil des Klassentests.
    - Über Klasseninstanzgrenzen hinweg:

      Hier können Attributwerte der anderen Klasseninstanzen auch Ein-/Ausgabezustände sein.



Modultest

- Was ist ein Modultest?
  - ⇨ Beim Modultest wird für ein Modul die Erfüllung der Spezifikation geprüft
  - ⇨ Da ein Modul meistens alleinstehend nicht lauffähig ist, wird zunächst eine Testumgebung erstellt, d.h. Programme, welche das Modul mit Testdaten versorgen
  - ⇨ Fehler können dann z.B. mit einen Debugger lokalisiert werden
- Wie führe ich einen Modultest durch?
  - ⇨ als Entwickler in einer simulierten Zielumgebung
  - ⇨ ohne den Auftraggeber
- Was wird in einem Modultest getestet?
  - ⇨ Die Funktion eines (einzelnen) Moduls

Der Modultest ist ein Black Box- oder White Box-Test
⇨ systematischer Entwicklertest



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.3 Integration und Integrationstest



Zur Erinnerung: Tätigkeiten im Design

- Zerlegung des Systems in Programmteile & Systemkomponenten ("Module")
  - ⇨ Anbindung an konkrete Programmiersprache, Betriebssystem, Hardware
  - ⇨ Ausarbeiten von Details
    - Klassen mit Methoden und Attributen mit Datentypen
    - Abläufe
    - Betriebssystemanbindung (Threads, Scheduler, Timer,…)
  - ⇨ Umsetzung der Vorgaben aus der Architektur

- Das Grob-Design (die SW-Architektur) zerlegt das System in Systemkomponenten
  - ⇨ Das Zusammenbauen von Systemkomponenten zu Teilsystemen (Sub-Systemen) heißt "**Integration**"
  - ⇨ Der Test, ob mehrere Systemkomponenten fehlerfrei zusammen wirken, heißt "**Integrationstest**"



Integrationstest – die Idee

- Ein komplexes System besteht aus (sehr) vielen Systemkomponenten, die zu unterschiedlichen Zeitpunkten fertig werden
- Das Lokalisieren von Fehlern in einem solchen System ist sehr schwer



Ich erweitere die Teilsysteme Schritt für Schritt und führe wiederholt Tests auf dem inkrementell wachsenden Teilsystem durch

- Ich mache Änderungen an nur wenigen Stellen
  - ⇨ neu entdeckte Fehler hängen mit der letzten Änderung zusammen
  - ⇨ Fehler können aber durchaus auch in den "alten" Teilen liegen

⇨ "Inkrementelle Integration" statt "Big-Bang-Integration"





Inkrementelle Integration

Außerdem: Sukzessives Ersetzen von Stubs und Drivers durch echte Module



Top-Down-Vorgehensweise

- Zuerst Implementierung, Montierung und Test der obersten Schicht,
  - ⇨ die noch nicht hinzugefügten Funktionen werden durch Stubs simuliert
  - ⇨ sukzessives Ersetzen der darunter liegenden Stubs durch echte Funktionen

Version 1

TOP
Modul 1
STUB 2
Modul 3
STUB 1.1
Modul 1.2
STUB 3.1
STUB 3.2

Version 2

TOP
Modul 1
Modul 2
Modul 3
Modul 1.1
Modul 1.2
STUB 3.1
STUB 3.2
STUB 1.1.1
STUB 1.1.2



Bottom-Up-Vorgehensweise

- Zuerst Implementierung, Montierung und Test der untersten Schicht,
  - ⇨ die darüber liegenden Module werden durch "Driver" simuliert
  - ⇨ sukzessives Ersetzen der darüber liegenden Driver durch echte Funktionen



Vergleich: Bottom-Up-Vorgehensweise ⇔Top-Down-Vorgehensweise

- Bottom-Up-Vorgehensweise
  - ⇨ Vorteil:
    - Ergebnisse die auf oberster montierter Schicht zu sehen sind, sind echt, da das System darunter schon fertig ist
    - Es kommen keine Stubs zur Anwendung
  - ⇨ Nachteil:
    - Der Benutzer sieht erst in den späten Phasen der Entwicklung etwas, das dem Endergebnis entspricht (Benutzschnittstelle ist normalerweise oben!).
- Top-Down-Vorgehensweise
  - ⇨ Vorteil:
    - Der Benutzer sieht schon zu Beginn der Integration etwas
  - ⇨ Nachteil:
    - Die Ergebnisse auf oberster Schicht sind nicht echt (Simulationen)
- Kombination von Bottom-Up und Top-Down ist möglich
  - ⇨ Es kommen dann gleichzeitig Drivers und Stubs zum Einsatz



Integrationstest

- Was ist ein Integrationstest?
  - ⇨ Prüfung, ob mehrere Module fehlerfrei zusammenwirken
  - ⇨ Test von Teilen des Gesamtsystems
  - ⇨ Fokus auf das Zusammenspiel der Systemkomponenten
- Wie führe ich einen Integrationstest durch?
  - ⇨ durch ein Test- bzw. Integrations-Team
  - ⇨ Schrittweiser Zusammenbau des Teilsystems und Ansteuerung mit Drivers und Verwendung von Stubs
- Was wird in einem Integrationstest getestet?
  - ⇨ Ein Verbund aus
    - bereits einzeln getesteten Systemkomponenten oder
    - bereits integrierte und getestete Teilsystemen
  - ⇨ Die verwendeten Module hängen von deren Verfügbarkeit zum Zeitpunkt des Tests, sowie der gewählten Integrationsstrategie ab

Ein Integrationstest ist ein Test für ein Teilsystem bestehend aus mehreren Systemteilen

⇨ sowohl White-Box als auch Black-Box-Tests

⇨ mit Fokus auf die Interaktion der integrierten Komponenten



Frage Integrationstest

- Wo finden wir im SW-Entwurf mit UML für den OO-Test die Aufrufhierarchie?
  - ⇨ In den Sequenzdiagrammen
- Wie sind Sie in im Praktikum vorgegangen?
- Wie würden Sie Vorgehen beim Test von Konstruktor Dosierverwalter
  (bzw. entsprechende Fkt. zur Erstellung der Dosierer in Rezeptprozessor)?
  Was ist der Vorzustand, die Eingabe, der Nachzustand, die Ausgabe?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.4 Systemtest, Abnahmetest



Unterschied zwischen Systemtest und Abnahmetest

Was ist ein Abnahmetest?

- die Abnahmetests sind Bestandteil des Vertrags
- Nach erfolgreicher Abnahme wird bezahlt

Wie führe ich einen Abnahmetest durch?

- Zusammen mit dem Auftragsgeber in der echten Zielumgebung
- Mit formeller Protokollierung

Was ist ein Systemtest?

- Test des integrierten Gesamtsystems
- Alles, was beim Abnahmetest getestet wird - aber noch ausführlicher

Wie führe ich einen Systemtest durch?

- ohne den Auftraggeber,
  mit Entwicklern und Qualitätssicherung in der echten Zielumgebung



Unterschied zwischen Systemtest und Abnahmetest

Was wird in einem Abnahmetest getestet?

- ⇨ Alles was spezifiziert wurde:
- ⇨ Funktionstests ✔
- ⇨ Leistungstests (z. B. Antwortzeiten bei Normallast)
- ⇨ Benutzbarkeitstests (z. B. einheitliche Benutzerschnittstelle, Help-Bildschirme, Benutzerhandbuch)
- ⇨ Robustheitstests (Prüfen der Reaktion bei unzulässigen Eingaben)
- ⇨ evtl. Volumentests (z. B. maximale Datenmengen)
- ⇨ evtl. Stresstests (z. B. maximale Anforderung von Prozessorleistung)

Der Abnahmetest ist ein Blackbox-Test für das Gesamtsystem

⇨ aus Auftraggeber-Sicht

Was wird in einem Systemtest getestet?

- ⇨ Funktionstests, Leistungstests, Benutzbarkeitstests, Robustheitstests Volumentests, Stresstests, (evtl. ausführlicher als beim Abnahmetest)
- ⇨ außerdem Sicherheit, Zuverlässigkeit, Wartbarkeit, Dokumentation, ...
- ⇨ auch Installation, Datenkonvertierung, Start und Initialisierung
- ⇨ auch Betriebsfunktionen wie Backup und Recovery
- ⇨ auch Provozieren von Abstürzen, um Lauffähigkeit und Robustheit zu prüfen (Datei defekt, Plattenausfall, Datenbank korrupt, …)
- ⇨ auch Berücksichtigung von außergewöhnlichen Zuständen des Systems (Datenbank etc.).
- ⇨ auch Berücksichtigung von Schnittstellen nach außen (andere IT-Systeme. Peripheriegeräte wie Drucker etc.).

Der Systemtest ist ein Blackbox-Test für das Gesamtsystem

⇨ aus Auftragnehmer-Sicht

Leistungs-(Performance-)Test

- Bedingungen
  - ⇨ "Normale Last"
  - ⇨ Messen von Antwortzeiten und Kapazität

- Selbstverständliche Forderungen:
  - ⇨ Sofortige Antwort bei Tastendruck
  - ⇨ 1-2 Sekunden für leichte Fragen
  - ⇨ Ein Bescheid, wenn es länger dauert
  - ⇨ Batch Jobs über Nacht fertig



Volumentest

- Ziele
  - ⇨ Maximale Datenmengen
  - ⇨ Mehr als Maximum
  - ⇨ Randomgenerierte oder alte Daten
  - ⇨ Versuchen Sie, den ganzen Platz zu belegen

- Beobachtungen
  - ⇨ Verliert das System Daten?
  - ⇨ Verfälscht das System Daten?
  - ⇨ Stoppt das System mit Meldung / ohne Meldung?
  - ⇨ Braucht es enorme Zeit?

Stresstest

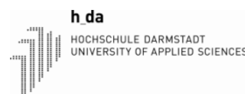
- Ziele
  - ⇨ Maximale Input-Geschwindigkeit
  - ⇨ Input an allen Linien (z. B. alle Sensoren senden gleichzeitig)
  - ⇨ Dasselbe Kommando an allen Linien
  - ⇨ Alarm an allen Linien
  - ⇨ Komplizierte Funktionen
  - ⇨ Nicht alle Systemressourcen zugänglich
- Reaktionen
  - ⇨ Werden Inputs vergessen?
  - ⇨ Bricht das System zusammen?
  - ⇨ Braucht es zu lange Zeit?
  - ⇨ Werden Daten verfälscht?
- Ein ungeheuer wichtiger Test bei Realzeitsystemen!

Benutzbarkeitstest

- Evaluation durch Experten (Ergonomen)
- Standard für Benutzerschnittstelle
- Messen der Lernkurve
- Beurteilung der Voraussetzungen

Sicherheitstest

- Ziel: Durchbrechen der Zugangskontrolle des Systems
- Kontrollieren Sie die Zugangsprivilegien für jede Benutzerklasse und jede Funktion
- Destruktiver Test
- Kontrolle von Backup- und Recovery-Funktionen
- Paralleler Zugriff von mehreren Terminals auf die selben Resourcen
- Virus Check
- Arbeitsumfeld (Organisation)



Weitere Tests

- Abhängig von den erwarteten und erwünschten Systemeigenschaften
  - ⇨ Kunde
    - Erweiterungsfähigkeit
    - Integrationsfähigkeit
    - Zuverlässigkeit
    - Robustheit, Fehlertoleranz
  - ⇨ Intern
    - Wartungsfreundlichkeit
    - Erweiterungsfähigkeit
    - Portabilität



Test-Gesamtplanung

- Erstellung der Montage- und Testplanung, Erstellung der Testfälle erfolgt bereits an den jeweiligen Phasenenden (V-Modell)



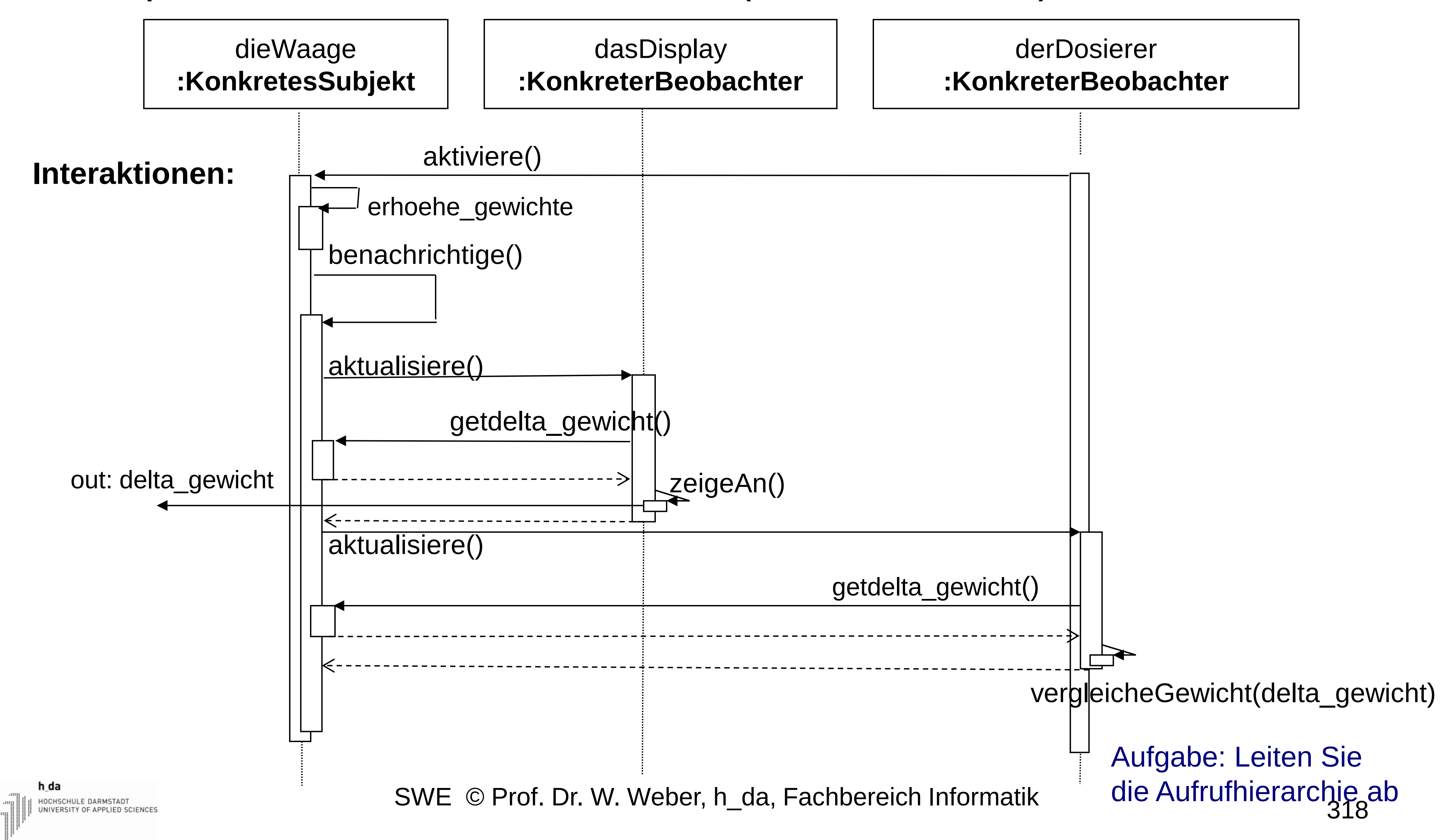
Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.5 Beispiel Modul-, Integrationstest



Beispiel Modultest - Beobachtermuster (Observer Pattern)



Aufgabe: Leiten Sie die Aufrufhierarchie ab

Beispiel Modultest Beobachtermuster (Observer Pattern)

Es sollen folgende Funktionen getestet werden:

- getdelta_gewicht von Waage (Stubs nicht erforderlich)
- vergleicheGewicht von Dosierer (Stubs nicht notwendig)
- aktualisiere von Dosierer (Modultest (mit Stubs und) mit schon ausprogrammierten aufzurufenden Funktionen)
- aktiviere von Waage (nur Test mit schon ausprogrammierten Funktionen)



Beispiel Modultest

Aufrufhierarchie in Form eines Strukturdiagramms für das Beispiel Observermuster der Praktikumsaufgabe

Woher kommt Delta-/Sollgewicht von Zutat?? Parameter? Attribut? Attribut von welcher Instanz?

Beispiel Modultest Beobachtermuster (Observer Pattern)

- Können Sie eine Funktion einer Klasse einfach aufrufen?
  - ⇨ Nein, erst instanziieren, initialisieren der Klasseninstanzen
- Welche Zustände müssen Sie setzen zum Austesten welcher Funktionen, was sind Parameter?
  - ⇨ Delta-Gewicht, zu dosierendes Sollgewicht gemäß Rezeptschritt, Erhöhungsgewicht? für welche Funktion? und zusätzlich??
- Wo erfolgt das Instanziieren, Setzen der Zustände, Übergeben, Annehmen der Parameter der zu testenden Funktion, Prüfung der Ergebnisse?
  - ⇨ Im Driver



Wie schreiben wir Tests?

- Wohin?
- Wie?



Wie schreiben wir Tests?

Wollen Sie es so realisieren (Tests oder Testaufrufe in main())?
Oder: Erstellung der Test-Funktionen in eigener Klasse (z. B. WaageTest)?

```
void main()
// includes
..
//Test-Driver:
void Waage_getdelta_gewichtTest ()
{
     //Instanziieren der Klasseninstanzen, Vorzustand setzen, d. h.  z. B. setze Attribut dieWaage.deltaGew zu 4
     // (Evtl. Instanziieren von Pointern auf Klassenobjekte global und Kreieren von Objekten zu Pointern und Setzen von Attributen
     //auch außerhalb in spezieller SetUp-Fkt., die vor jedem Test aufgerufen wird)
Waage dieWaage;
dieWaage.SetDeltaGew(4);
     //Aufruf von dieWaage.getdelta_gewicht() und
     //Testen des Ergebnisses auf Korrektheit (per assert-Statement) oder Ausgabe der Ergebnisse (anschließend:
     // manueller Vergleich)
assert (dieWaage.getdelta_gewicht()==4); //oder if ...
     //Allgemein: Ergebnis kann sein:
     //Rückgabe-Parameter / &-Parameter oder
     //Zustände von Attributen, globale Variablen, Datei- / Datenbankfelder (Nachzustand)

    //Zurücksetzen Attribute / destruieren Klasseninstanzen evtl. auch in spezieller tearDown-Funktion, die nach jedem Test
    //aufgerufen wird (z. B. durch new erzeugte Instanzen müssen gelöscht werden.)
}
```



Wie schreiben wir Tests?

```
void Dosierer_vergleicheGewichtTest (deltaGewicht) // sollGewicht als Attribut in Dosierer?
{
. . .
. . .
}

void Dosierer_aktualisiereTest ()
{
    //Instanziieren der Klasseninstanzen
    //setze Attribut ...
       ..
    //Aufruf von aktualisiere() ... //(1. mit getdelta_gewicht() und vergleicheGewicht() als Stubs bzw.)
                  //2. mit getdelta_gewicht() und vergleicheGewicht() als schon fertige Funktionen (Integrierte Funktionen)
       ..
    //Ausgabe Ergebnis
                  ..
     //Zurücksetzen Attribute
}

void Dosierer_aktualisiereTest2 () ...  // evtl. nur mit Stubs oder mit anderen Rückgabewert für gew_Kennz
```



Wie schreiben wir Tests?

```
// eigentlicher Test
//Test 1:
..... Waage_getdelta_gewichtTest ();
//Test 2:
.....Dosierer_vergleicheGewichtTest ();
//Test3
.....Dosierer_aktualisiereTest ();
//Test 4:
.....Dosierer_aktualisiereTest2 ();  //mit Stubs
..
//Test n:
.....


..
```

Beispiel Integrationstest

- Teilsysteme d, e ... werden zu größeren Teilsystem c zusammengesetzt.
  (d oder e kann auch Fremdsystem sein.)
- Beispiel in unserem Praktikum:
  d und e sind Teilsysteme mischen und start,
  c ist Gesamtsystem
  => Integrationstest der obersten Stufe
  (bei uns recht trivial)
  => Ein Testfall für diesen Integrationstest ist der normalerweise schon nach der Anforderungsdefinition spezifizierte Abnahmetestfall



Beispiel Integrationstest

Das muss alles beim Schreiben des Testfalls berücksichtigt werden

Teil- bzw. Gesamtsystem

Teilsystem start

Teilsystem mischen

Gewichtwert von externer Waage (wenn simuliert => deterministisch!)

Kommandos Ventil auf, … an externe Dosierer, …

Daten von externen Subsystemen / Dateien? (Vorzustand)

Bildschirm-ausgabe B1, … ?

Bildschirm-eingabe B1, … ?

Attribute A1,… ? (Nachzustand)

Attribut A1, A2,… ? (Vorzustand)

Bildschirm-eingabe B2, … ?

Gew-Wert auf Display …



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

8.6 Testwerkzeuge

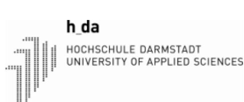


Testen in größeren Projekten (I)

- Stellen Sie sich nun vor, alle Ihre zu testenden Programme sind Module, die zu einem größeren Projekt gehören
  - ⇨ Die Module sollen regelmäßig integriert und getestet werden
    - Sie müssen die Testaufrufe aus den Main-Funktionen der Einzelprojekte zu EINER - zunehmend unübersichtlichen - Main-Funktion zusammenführen
    - Die Tests für ein Modul können dann nicht mehr einzeln durchgeführt werden
    - Jeder neue Testfall erfordert eine Änderung an der Main-Funktion.
    - Die Reihenfolge der Tests kann evtl. eine Auswirkung auf das Ergebnis haben
  - ⇨ Die manuelle Durchführung dieser Tests
    - wäre in der Summe sehr aufwändig
    - wäre nicht reproduzierbar
    - wäre nicht systematisch
    - wäre nicht zuverlässig genug für ein professionelles Projekt



In einem größeren Projekt ist der einfache Test-Ansatz über eine Main-Funktion nicht brauchbar!



Testen in größeren Projekten (II)

- Üblicherweise findet in einem größeren Projekt die Entwicklung von Modulen in verschiedenen Teams statt
  - ⇨ Die Tests müssen einzeln durchführbar sein – aber auch leicht zusammen zu führen sein
  - ⇨ Jedes Team entwickelt "seine" Klassen und schreibt entsprechende Tests
- Üblicherweise findet in einem größeren Projekt die Entwicklung in Iterationen statt
  - ⇨ Nach jeder Iteration - also täglich oder mindestens wöchentlich - muss getestet werden
    - ob die entwickelten Module (noch) die Modultests besteht
    - ob die integrierten Module die Integrationstests bestehen



In einem größeren Projekt müssen Tests
- ⇨ verteilt und unabhängig voneinander entwickelt werden
- ⇨ automatisiert durchgeführt werden



xUnit

http://cppunit.sourceforge.net/projects/cppunit

- xUnit = Familie von Open Source Frameworks für das Testen
  - JUnit für Java
  - CppUnit für C++
  - NUnit für .Net
  - uvm. ...
  - ⇨ alle mit gleichem Funktionsprinzip und ähnlicher Verwendungsweise
- xUnit ermöglicht
  - ⇨ die verteilte und unabhängige Entwicklung von funktionalen Tests
  - ⇨ die einfache Zusammenführung der verschiedenen Tests
  - ⇨ die automatische Durchführung der Tests
  - ⇨ die automatische Protokollierung und Überprüfung der Ergebnisse
  - ⇨ Tests in der gleichen Programmier-Sprache wie die Anwendung
  - ⇨ und viele anderer nützliche Dinge...

xUnit ist die Standardlösung
für (open Domain) Testframeworks !



Ein einfacher Test mit CppUnit (I) (Prinzip) – Aufgabe

- Wir wollen eine Klasse Complex zum Rechnen mit komplexen Zahlen in C++ entwickeln
  - ⇨ mit den üblichen mathematischen Operatoren
- Wir wollen eine Testklasse ComplexNumberTestCase entwickeln, die die Klasse Complex mit CppUnit testet
  - ⇨ Die Testklasse enthält
    - Eine Testmethode (runTest), welche Methoden der zu testenden Klasse aufruft
    - Die Methoden setUp() und tearDown() die vor bzw. nach jeder Testmethode ausgeführt werden
  - ⇨ Mit speziellen Zusicherungen ("Assert's") werden zu prüfende "Soll-Ergebnisse" festgelegt
- Bei der testgetriebenen Entwicklung entwickeln Sie erst den Test und dann die zu testende Klasse

Die zu testende Klasse wird nicht modifiziert !

C++-Unit - Ein einfacher Test-Case (Prinzip)

Als Erstes entwickeln wir den Testfall (Testgetriebene Entwicklung):

```
#include ... // hier müssen u. a. Dateien des Frameworks inkludiert werden
class ComplexNumberTestCase : public CppUnit_NS::TestCase
{
  public:
    ComplexNumberTest(string name ) : CppUnit_NS:: TestCase( name ) {}
     void runTest() {
        CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
        CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) ); }
};
```

Testmethode

Zusicherungen
(falls falsch=> Fehler)

Wir entwickeln nun die zu testende Klasse Complex und den Operator ==, so dass der Test läuft:

```
class Complex {
  friend bool operator ==(const Complex& a, const Complex& b);
  double real, imaginary;
public:
  Complex( double r, double i = 0 )
    : real(r), imaginary(i) { }
};
bool operator ==( const Complex &a, const Complex &b ){
    return a.real == b.real  &&  a.imaginary == b.imaginary;
};
```

Dann können wir den TestCase ComplexNumberTestCase instanziieren und runTest() aufrufen

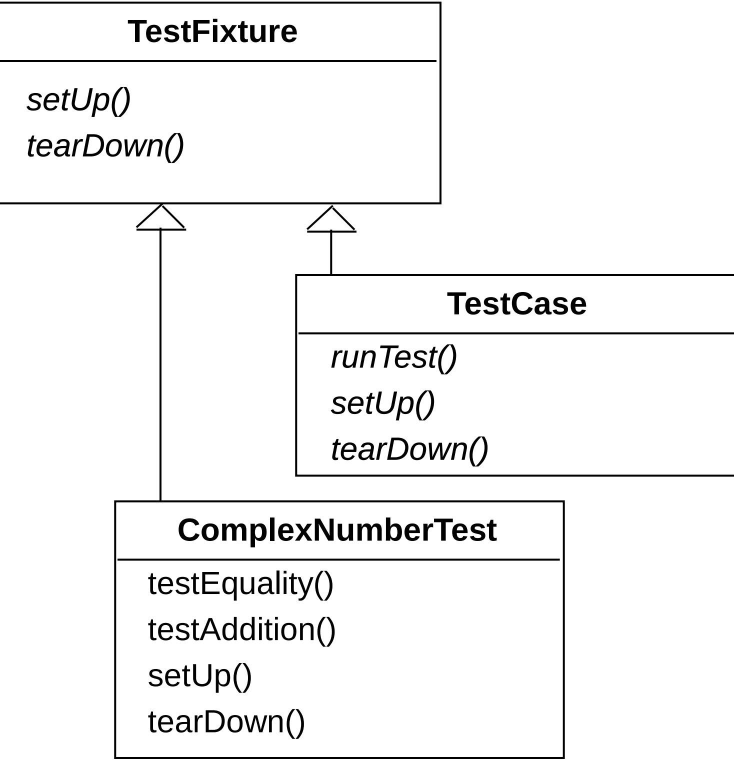


C++-Unit - Mehrere Tests mit setUp und tearDown - TestFixture

- Normalerweise wollen wir einige Instanzen der zu testenden und evtl. weiterer Klassen incl. Daten vorher definieren und
- mit diesen Instanzen eine Anzahl von Tests fahren (z. B. testEquality() (s. o.) und TestAddition()). Bei jedem Test soll aber der Zustand vor dem jeweiligen Test jeweils in den definierten Anfangszustand gebracht werden.

⇒ Wir verwenden die Klasse TestFixture, in der die abstrakten Operationen *setUp* und *tearDown* stehen, die vor/nach jedem Test den definierten Zustand herstellen.

⇒ die setUp und tearDown-Methoden wie auch die eigentlichen Tests (z. B. testEquality und testAddition) werden in der eigentlichen Testfixture (z. B. ComplexNumberTest) implementiert. Die eigentlichen TestCases werden dann durch das C++-Unit-Framework zusammengestellt und aufgerufen.





C++-Unit - setUp und tearDown in der TestFixture

```
#include ...
class ComplexNumberTest : public CppUnit_NS::TestFixture
{
private:
  Complex *m_10_1, *m_1_1, *m_11_2;
public:
  void setUp()
  {
    m_10_1 = new Complex( 10, 1 );
    m_1_1 = new Complex( 1, 1 );
    m_11_2 = new Complex( 11, 2 );
  }
  void tearDown()
  {
    delete m_10_1;
    delete m_1_1;
    delete m_11_2;
  }
};
```

Members zum Testen

Testvorbereitung

Testnachbereitung

Was muss jetzt noch hinzugefügt werden?



C++-Unit - setUp und tearDown und die einzelnen Tests in der TestFixture

```
#include ...
class ComplexNumberTest : public CppUnit_NS::TestFixture
{
private:
  Complex *m_10_1, *m_1_1, *m_11_2;
public:
  void setUp()
  {
    m_10_1 = new Complex( 10, 1 );
    m_1_1 = new Complex( 1, 1 );
    m_11_2 = new Complex( 11, 2 );
  }
  void tearDown()
  {
    delete m_10_1;
    delete m_1_1;
    delete m_11_2;
  }
  void testEquality()
  {
    CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
    CPPUNIT_ASSERT( !(*m_10_1 == *m_11_2) );
  }
  void testAddition()
  {
    CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
  }
};
```

Testmethoden

Zusicherungen

Zu jeder der hier aufgeführten 2 Testmethoden wird vom Framework ein TestCase mit einem runTest() erstellt, in dem jeweils setUp() und tearDown() vor und nach dem eigentlichen Test (hier z. B. testEquality) abläuft.



C++-Unit mit TestSuite - Überblick

* Darstellung vereinfacht. Tatsächlich gibt es deutlich mehr Klassen und Methoden innerhalb des Frameworks.

C++-Unit mit TestSuite

- Wie konstruiere ich den Test, so dass ich alle Tests zusammen laufen lassen kann?

=> Kreiere eine Suite mit mehreren Tests:

- ⇨ Mit Hilfe des Frameworks werden einzelne TestCases aus den Methoden der konkretenTestfixture (z. B. ComplexNumberTest) gebildet und der Suite hinzugefügt. Jeder TestCase besteht aus der SetUp-, der Test-, der TearDown- und der run(TestResult*)-Methode. In der run(TestResult*)-Methode werden die anderen Methoden aufgerufen.
- ⇨ Beim Ausführen der Suite werden dann alle Elemente der Suite ausgeführt (run(...) aller Elemente wird aufgerufen)
- ⇨ Elemente der Suite können TestCases oder auch Suites sein (Compositum-Muster!)
- ⇨ Falls das Element ein TestCase ist, wird durch Run(...) die SetUp-, Test- und TearDown-Methode des TestCase ausgeführt
- ⇨ Falls das Element eine TestSuite ist, werden in run(...) die Run()-Methoden der darunterliegenden Elemente aufgerufen
- ⇨ Die oben aufgeführten Schritte und die Registrierung der Suite werden über Macro-Aufrufe des Frameworks durchgeführt

C++-Unit mit TestSuite – Erstellung und Registration in der TestFixture

```
#include ...
class ComplexNumberTest : public CppUnit_NS::TestFixture
{
  CPPUNIT_TEST_SUITE(ComplexNumberTest);
  CPPUNIT_TEST(testEquality);
  CPPUNIT_TEST(testAddition);
  CPPUNIT_TEST_SUITE_END();
private:
  Complex *m_10_1, *m_1_1, *m_11_2;
public:
  void setUp()
  {
    m_10_1 = new Complex( 10, 1 );
    m_1_1 = new Complex( 1, 1 );
    m_11_2 = new Complex( 11, 2 );
  };
  void tearDown()
  {
    delete m_10_1;
    delete m_1_1;
    delete m_11_2;
  };
  void testEquality()
  {
    CPPUNIT_ASSERT( *m_10_1 == *m_10_1 );
    CPPUNIT_ASSERT( !(*m_10_1 == *m_11_2) );
  };
  void testAddition()
  {
    CPPUNIT_ASSERT( *m_10_1 + *m_1_1 == *m_11_2 );
  };
};
CPPUNIT_TEST_SUITE_REGISTRATION(ComplexNumberTest);
```

Aufbau einer Testsuite aus einzelnen Testfällen

ein TestCase testEquality mit den Methoden setUp(), testEquality() und TearDown() wird erstellt und in der Suite ComplexNumberTest hinzugefügt.

Registrierung der Testsuite in der Registry des Frameworks

C++-Unit - Der Testlauf

- Zur Ausführung des Tests muss aus den Angaben in der Registry der Test erstellt und anschließend ausgeführt werden:

```
#include ....
int main()
{
  CppUnit_NS::TextUi::TestRunner runner;
  runner.addTest(CppUnit_NS::TestFactoryRegistry::getRegistry().makeTest() );
  bool wasSuccessful = runner.run();
  return wasSuccessful;
}
```

makeTest() erstellt aus den Angaben der Registry den Gesamttest

- Im Framework gibt es noch Listener, Resultcollectors und Outputter mit denen die Results des Tests gesammelt und übersichtlich ausgegeben werden können



Testen mit xUnit – Das Ergebnis

- In den Zusicherungen wird das Soll-Ergebnis mit dem Ist-Ergebnis verglichen
  - ⇨ Gibt es einen Unterschied, erfolgt eine entsprechende Meldung
  - ⇨ Die Tests werden jedenfalls weiter ausgeführt (und nicht abgebrochen)

Problems | Tasks | Console | Properties | Debug

```
<terminated> Complex.exe [C/C++ Local Application] C:\Links\SWE\Development\Complex\Debug\Complex.exe
ComplexNumberTest::testEquality : OK
ComplexNumberTest::testNonEquality : OK
ComplexNumberTest::testAddition : assertion
ComplexNumberTest::testDivision : OK
ComplexNumberTest::testDivideByZeroThrows : OK

complexTest.cpp:49:Assertion
Test name: ComplexNumberTest::testAddition
assertion failed
- Expression: *m_10_1 + *m_1_1 == *m_11_2

Failures !!!
Run: 5   Failure total: 1   Failures: 1   Errors: 0
```

In der Methode *testAddition* wurde eine Zusicherung verletzt!

Mit xUnit können Tests
- ⇨ verteilt und unabhängig voneinander entwickelt werden
- ⇨ automatisiert durchgeführt werden



Testen mit xUnit – Sonstiges

- Im xUnit Framework
  - ⇨ ist es recht einfach Tests zu erstellen, wenn man das Prinzip verstanden hat. Ein funktionierendes Beispiel erleichtert den Start aber erheblich!
  - ⇨ ist der Zugriff auf private Attribute und Methoden normalerweise nicht möglich. Es gibt aber oft "Tricks" um zu Testzwecken darauf zugreifen zu können
  - ⇨ wird der praktische Einsatz diverser Design-Patterns demonstriert
  - ⇨ kann testgetrieben entwickelt werden. d.h. es wird immer erst ein Test geschrieben und erst dann der Code, der den Test zum Erfolg bringt.

Testen mit xUnit – im Gesamtzusammenhang

- Mit entsprechenden Addons zu xUnit
  - ⇨ kann aus den Testfällen eine Dokumentation der Testfälle erzeugt werden
  - ⇨ kann die erzielte Anweisungs- oder Zweigüberdeckung berechnet und dargestellt werden ("Test Coverage")
  - ⇨ können Messungen für Last- und Performancetests ("Profiling") durchgeführt werden
  - ⇨ können leere Rahmen für zu schreibende Testklassen erzeugt werden
  - ⇨ ...
- Es gibt neben den Unit-Tests weitere Testwerkzeuge
  - ⇨ für die Verwaltung, Planung und Automatisierung der Testdurchführung
  - ⇨ für automatisierte GUI-Tests
  - ⇨ zur Durchführung von "statischen Tests"
    (die den Code "nur" analysieren, aber nicht ausführen)
  - ⇨ uvm...



Literatur zu Testen

Balzert, Helmut: Lehrbuch der Softwaretechnik: Softwaremanagement, Softwarequalitätssicherung, Unternehmensmodellierung; Spektrum, akademischer Verlag; 1998

Vigenschow, Uwe: Objektorientiertes Testen und Testautomatisierung; dpunkt-Verlag; 2005

Westphal, Frank: Testgetriebene Entwicklung mit JUnit & FIT; dpunkt-Verlag; 2006

Köhler: Der C/C++ Projektbegleiter; dpunkt Verlag; 2007

http://cppunit.sourceforge.net/projects/cppunit

http://sourceforge.net/apps/mediawiki/cppunit

http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html

http://www.evocomp.de/tutorials/tutorium_cppunit/howto_tutorial_cppunit.html



Kontrollfragen Test und Integration

- Wann ist ein Programm fehlerfrei?
- Was bedeutet es, wenn alle Tests fehlerfrei laufen
- Ist es normalerweise möglich alle möglichenTestfälle laufen zu lassen?
- Welche Prinzipien zur Auswahl von Testfällen kennen Sie?
- Wie unterscheiden sich Black-Box-Tests und White-Box-Tests?
- Welche Unterschiede gibt es zwischen Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung?
- Was ist Integration?
- Was halten Sie von der Aussage "Sie haben das Programm doch getestet – wie kann dann so ein Fehler auftreten?"
- Warum muss man in der Architektur, im Design,… schon an Test denken?
- Warum gibt es im V-Modell 4 Testphasen? Wie unterscheiden sie sich?
- Woher "weiß" CppUnit welche Tests ausgeführt werden sollen, wenn die Testklassen nicht in der Main-Funktion instanziiert werden?

Können Sie jetzt Tests spezifizieren, entwickeln und anwenden?



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

Teil II - Querschnittsthemen



Was sind Querschnittsthemen?

- Bisher wurden die Themen entlang eines Projektdurchlaufs behandelt
  - ⇨ es gibt aber auch Themen, die den einzelnen Phasen übergeordnet sind
  - ⇨ oder in vielen Phasen vorkommen

- Wir behandeln folgende Querschnittsthemen (weiterhin aus der Sicht eines Software-Entwicklers):
  - ⇨ Qualitätssicherung
  - ⇨ Software-Metriken
  - ⇨ Vorgehens- und Prozessmodelle
  - ⇨ Technisches Management
  - ⇨ Prozessorientiertes Qualitätsmgmt (ISO, CMM, Assessments)
  - ⇨ ... es gibt aber noch viele andere Themen



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

9. Qualitätssicherung



Qualitätssicherung

- Qualitätssicherung beinhaltet Maßnahmen, die sicherstellen sollen, dass ein Produkt ein vorgegebenes Qualitätsniveau erreicht
  - ⇨ es geht darum, eine **vorgegebene** Qualität zu erreichen - nicht unbedingt um die höchste Qualität

- **Produktqualität**
  - ⇨ Die Maßnahmen zielen auf die Qualität der Bestandteile des Produkts.
    - Anwendung **konstruktiver** Maßnahmen bei der Entwicklung des Produktes
    - Anwendung **analytischer** Maßnahmen zur Untersuchung der Produktteile

siehe nächste Folien !

- **Prozessqualität**
  - ⇨ Die Maßnahmen zielen auf die Qualität des Prozesses mit dem das Produkt erstellt wird.
    - Definition bzw. Festlegung und Anwendung eines Entwicklungsprozesses,
    - Aktive Suche nach Schwachstellen im Entwicklungsprozess,
    - Systematische Verbesserung des Entwicklungsprozesses (so dass beim nächsten Projekt nicht wieder die gleichen Fehler gemacht werden)
    - siehe z.B. Norm ISO 9001; CMM, CMMI; SPICE

Produktqualität: Konstruktive Qualitätssicherung

- Es wird versucht, das Entstehen von Qualitätsmängeln bereits während der Entwicklung (**Konstruktion**) des Produkts zu verhindern!
  - ⇨ Beseitige bei entdeckten Mängeln nicht nur den Mangel selbst, sondern auch seine Ursache(n)
  - ⇨ Gestalte den Entwicklungsprozess so, dass Qualitätsmängel seltener werden
- Dazu werden während der Entwicklung geeignete Maßnahmen eingesetzt
  - ⇨ bewährte Methoden, Techniken, Konstruktionsprinzipien (UML, Pair Programming, Test First, OO-Entwurf, 4-Schichten-Architektur, ...)
  - ⇨ formale Verfahren (formale Spezifikation)
  - ⇨ Werkzeuge (Case Tool, IDE, ...)
  - ⇨ Vorgehensmodelle und Vorgehensrichtlinien (Wasserfallmodell, V-Modell, Einsatz von Prototypen, Coding Guidelines, ...)

Qualität kann nicht nachträglich in ein Produkt "hineingeprüft" werden, Sie muss von Anfang an "eingebaut" werden!



Produktqualität: Analytische Qualitätssicherung

- Es wird versucht, entstandene Fehler zu entdecken (und zu beseitigen), indem die vorliegenden Teile des Produkts **analysiert** werden!
  - ⇨ Untersuche (Teil-)Produkte nach ihrer Fertigstellung auf Qualität
  - ⇨ Bessere nach, wo Mängel auftreten
- In der analytischen Qualitätssicherung unterscheidet man
  - ⇨ **dynamische Maßnahmen**, welche die "laufende" Software untersuchen
    - Tests: Performancetests, Robustheitstests, Testüberdeckungen usw. (vgl. Kap. Test)
  - ⇨ **statische Maßnahmen**, die eingesetzt werden können bevor die Software "läuft"
    - Manuelle Prüfverfahren zur Prüfung der fertigen Artifakte (= Reviews, vgl. OOAD)
    - Automatische Prüfverfahren zur Prüfung der fertigen Artifakte
      - Automatische statische Codeanalyse zur Überprüfung des Codes auf potenziell fehlerhafte Konstrukte (z.B. „=“ in einer If Bedingung) und auf Verstöße gegen die Implementierungsrichtlinien
      - Konsistenz-Checks über dem UML-Modell mit Hilfe des CASE-Tools
      - Kompilierung des Codes zum Finden formaler Fehler (z. B. Syntaxfehler)
      - Codeanalyse mit Werkzeugen zur Bestimmung von Qualitäts-Maßen (vgl. Kapitel Metriken)
    - Mathematisches Beweisen der Korrektheit von Programmen (Formale Verifikation)



QS-Maßnahmen - Übersicht

- QS-Maßnahmen
  - Sicherung Produkt-Qualität
    - konstruktiv QS
    - analytische QS
      - dynamisch — Test (Kap.8)
      - statisch
        - man. Prüfverf (Review)
        - automatische Prüfverf.
        - math. Beweisen
  - Sicherung Prozess-Qualität (Kap. 11)



Statische Codeanalyse (Beispiel einer analytischen statischen QS-Maßnahme)

- Bei der autom. statischen Codeanalyse untersucht ein Tool den Quellcode
  - ⇨ es werden "beliebte" Fehlerquellen entdeckt und angezeigt
    - Arrayüberschreitungen
    - Zuweisung statt Vergleich
    - Gefahr von Rundungsfehlern
    - uvm.
  - ⇨ z.B. „Codecheck“ in „Understand“, „QA-C++" von QA Systems oder „PC-lint for C/C++"
  - ⇨ Nette Übung bei PC-lint – suchen Sie den Fehler im "Bug of the month" http://www.gimpel.com/html/bugs.htm
  - ⇨ Sie können dort auch in der "Interactive Demo" Ihren eigenen Code untersuchen lassen

```
#include <iostream.h>

int main() {
      const double three = 3.0;
      double x, y, z;
      x = 1 / three;
      y = 4 / three;
      z = 5 / three;
      if( x + y == z )
          cout << "1/3 + 4/3 == 5/3 \n";
      else
          cout << "1/3 + 4/3 != 5/3 \n";
      return 0;
}
```

PC-lint liefert:

```
if( x + y == z ) bug777.cpp(15):
        Testing floats for equality
```

PC-lint weist auf ein (wahrscheinlich) unerwartetes Ergebnis des Vergleichs hin – nämlich *false* wegen Rundungsfehlern, die bei *float's* auftreten.



Formale Verifikation (Beispiel einer analytischen statischen QS-Maßnahme)

- Es wird mathematisch bewiesen,
  - ⇨ dass ein Programm zu zulässigen Eingabewerten die richtigen Ergebnisse (Ausgabewerte) liefert, d.h. richtig ist und
  - ⇨ dass der Algorithmus in jedem Fall terminiert.
- Vorgehen bei der formalen Verifikation
  - ⇨ Jede Funktionalität eines Softwaresystems hat die Aufgabe,
    - aus bestimmten Anfangsbedingungen über Daten („precondition“)
    - bestimmte Endbedingungen über Daten („postcondition“) herbeizuführen.
  - ⇨ Dies ist wie ein Vertrag zu sehen zwischen demjenigen,
    - der eine Funktionalität benutzt,
    - und dem, der sie erfüllt.
- Die formale Verifikation eines Systems verläuft nun folgendermaßen:
  - ⇨ Durch formales Anwenden von Regeln der mathematischen Prädikatenlogik wird über die Wirkungsweise des Algorithmus der Beweis geführt, dass er aus seinen Anfangsbedingungen die Endbedingungen herstellt.
  - ⇨ Gelingt dieser Beweis, erfüllt das System exakt die Spezifikation und ist korrekt.



Kontrollfragen Qualitätssicherung

- Was ist der Unterschied zwischen Prozessqualität und Produktqualität?
- Angenommen, Sie verwenden einen Standard zur Sicherung der Prozessqualität (z.B. ISO 9001). Entsteht dadurch eine gute Produktqualität?
- Angenommen, Sie unternehmen zur Sicherung der konstruktiven Produktqualität erhebliche Anstrengungen. Entsteht dadurch eine gute Produktqualität?
- Angenommen, Sie unternehmen zur Sicherung der analytischen Produktqualität erhebliche Anstrengungen. Entsteht dadurch eine gute Produktqualität?

Um gesicherte Produktqualität zu erzielen, müssen Sie alle Maßnahmen gezielt kombinieren!



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

10. Metriken



Motivation (I)

- Situation
  - ⇨ Projekt mit <u>100 Mitarbeitern</u>, <u>Laufzeit</u> <u>3 Jahre</u>
  - ⇨ anschließend <u>10 Jahre Wartung</u>
  - ⇨ SIE sind <u>Projektleiter</u>
- <u>Woher</u> <u>wissen</u> <u>Sie</u> (als Projektleiter) <u>frühzeitig</u>,
  - ⇨ ob das Ergebnis <u>wartbar</u> sein wird
  - ⇨ ob die <u>versprochene Funktionalität</u> geboten wird
  - ⇨ ob der <u>Code effizient</u> läuft
  - ⇨ ob die <u>Tests einfach erstellt werden können</u>
  - ⇨ ob das <u>Projekt aus dem Ruder</u> <u>läuft</u>
  - ⇨ ...
  - ⇨ ob die Investition von über 50 Mio. Euro ein <u>Erfolg wird</u>???



Kann man <u>aus Arbeitsergebnissen</u> (automatisch)
<u>Indizien</u> <u>für diese Dinge</u> <u>berechnen!?</u>
⇨ "Messungen" und "Metriken"

Motivation (II)

- Situation
  - ⇨ SIE sind einer der 100 Mitarbeiter
  - ⇨ Sie entwickeln selbständig einen Teil
- Woher wissen Sie vor dem Review,
  - ⇨ ob Ihr Arbeitsergebnis gut (genug) ist
  - ⇨ ob Sie genug Kommentare haben
  - ⇨ ob Ihr Code erweiterbar ist
  - ⇨ ob Ihr Code wiederverwendbar ist
  - ⇨ ob Ihr Code testbar ist
  - ⇨ wo Sie Ihr Ergebnis noch verbessern sollten
  - ⇨ ...
  - ⇨ ob Sie ein guter oder schlechter Entwickler sind???





Kann man aus Arbeitsergebnissen (automatisch)
Indizien für diese Dinge berechnen!?
⇨ "Messungen" und "Metriken"

Lernziel Metriken

- Sie sollen in diesen Kapitel lernen,
  - ⇨ wozu Metriken gedacht sind
  - ⇨ was mit Metriken gemessen werden soll
  - ⇨ was mit Metriken tatsächlich gemessen wird
  - ⇨ welche Standard-Metriken es gibt
  - ⇨ wie Tools den Umgang mit Metriken erleichtern
  - ⇨ wie Metriken sinnvoll in Projekten eingesetzt werden
  - ⇨ welche Gefahren Metriken bergen

Anschließend können Sie in einer vernünftigen Weise mit Metriken umgehen?!

Was will man messen?

- Arbeitsfortschritt und Qualitätseigenschaften

Aber wie kann man das messen??



Metriken – die Grundidee

- Leite elementare Eigenschaften aus den Arbeitsergebnissen ab:
  - ⇨ z.B. für Code
    - Codegröße (Anzahl der Zeichen, Zeilen etc.)
    - Anteil an Kommentaren
    - Anzahl an komplexen Konstrukten (IF, WHILE,...)
    - Anzahl von Vererbungen
    - ...
  - ⇨ z.B. für die Architektur
    - Anzahl der Abhängigkeiten zwischen Klassen
    - Größe der Dokumentation
    - ...
- Durch Kombination von elementaren Eigenschaften können Aussagen über SW-Qualitätseigenschaften abgeleitet werden z.B.
  - ⇨ Verständlichkeit
  - ⇨ Wartbarkeit
  - ⇨ ...



Es gibt aber viele Interpretationsmöglichkeiten...



Metriken: Einfache Ansätze

- Wie messe ich die Größe eines Programms (einer Funktion)?
  - ⇨ Anzahl Zeilen (LOC)
    - Durch viele Anweisungen in einer Zeile wird ein Programm nicht kleiner (einfacher)!
    - Durch Leerzeilen oder übertriebene Aufteilung aber auch nicht größer!
  - ⇨ Anzahl Anweisungen
    - Durch Zusammenfassung von Einzelanweisungen zu komplizierten Anweisungen wird ein Programm nicht kleiner!
  - ⇨ Anzahl Bytes
    - Durch Sparen von Bytes (z. B. kurze Variablennamen) wird ein Programm auch nicht kleiner!

Zähle nur die verschiedenen Operanden und die Operatoren
⇨ "Halstead-Metriken"



Halstead-Metriken: "Textuelle Komplexität"

- Anzahl der unterschiedlichen Operatoren: η1
- Anzahl der unterschiedlichen Operanden: η2
- Anzahl des Auftretens von Operatoren: N1
- Anzahl des Auftretens von Operanden: N2

Primitives Beispiel zur Veranschaulichung des Maßes:

```
void tausch(float &x, float &y)
{
        float z;
        z=x;
        x=y;
        y=z;
}
```

Operatoren	Anz.	Operanden	Anz.
void	**1**	**x**	**3**
( )	**1**	**y**	**3**
&	**2**	**z**	**3**
float	**3**		
=	**3**		
{ }	**1**		
,	**1**		
;	**4**		
η1 = 8	N1 = 16	η2 = 3	N2 = 9



Halstead-Metriken: "Textuelle Komplexität"

- Abgeleitete Maße:
  - ⇨ Größe des Vokabulars: $\eta = \eta 1 + \eta 2$
    - tausch: 8+3 = 11
  - ⇨ Länge der Implementierung: $N = N1 + N2$
    - tausch: 16 + 9 = 25
  - ⇨ Berechnete Länge: $N\hat{} = \eta 1 * ld(\eta 1) + \eta 2 * ld(\eta 2)$
    - tausch: N^ = 8*ld(8) + 3*ld(3)
      N^ = 24 + 4.75 = 28.75
  - ⇨ Programmgröße (Volumen): $V = N * ld(\eta)$
    - V = 25*ld(11) = 25 * 3,5 = 87,5
  - ⇨ es gibt viele weitere Maße mit umstrittener Aussagekraft...

Einfache Zuweisungen zählen genauso wie komplizierte Schleifen!?
⇨ "Strukturelle Komplexität": "McCabe Maß"

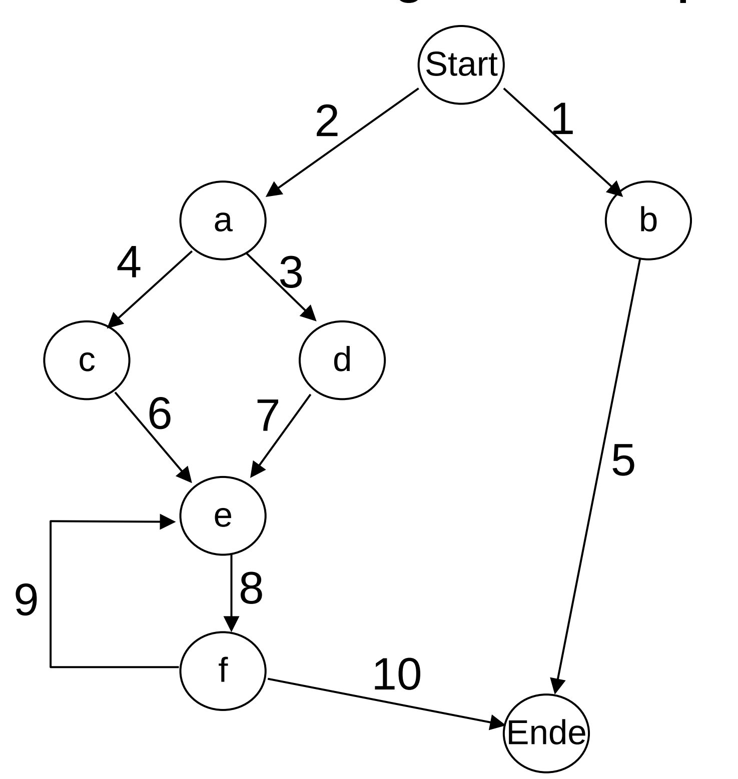
McCabe-Maß, "zyklomatische Zahl„: Strukturelle Komplexität

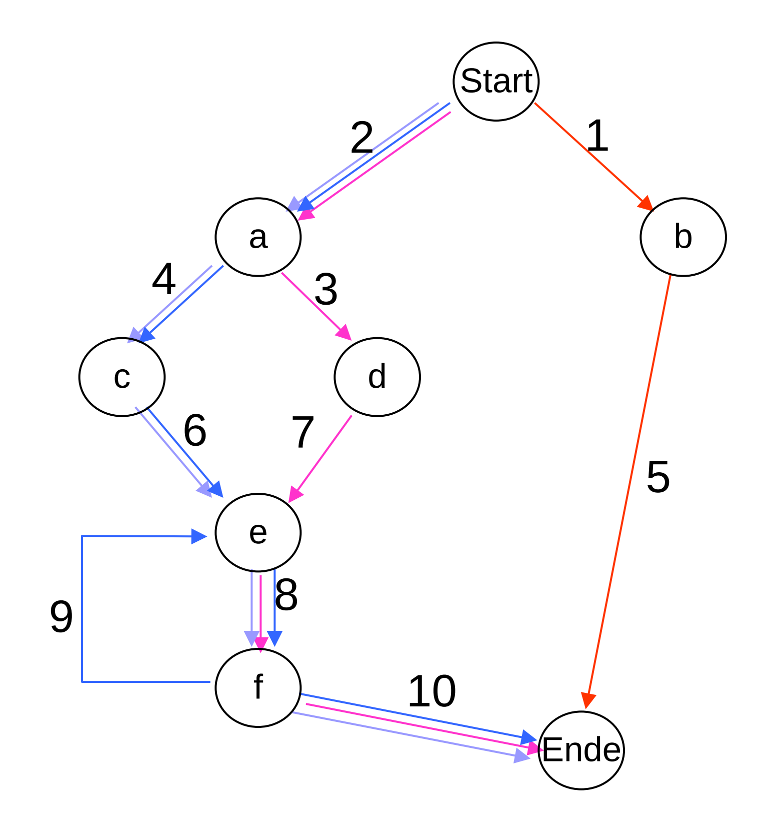
- Gibt Auskunft über die Komplexität der Kontrollstruktur einer Funktion

- zyklomatische Zahl z(G):
  - ⇨ Originaldefinition: z(G) = P
    - P = Anzahl der linear unabhängigen Pfade durch den Programm-Graphen G
  - ⇨ Alternative Berechnung z(G) = e – n + 2p
    - e: Anzahl Kanten von G,
    - n: Anzahl Knoten von G,
    - p: Anzahl der verbundenen Komponenten
  - ⇨ Alternative Berechnung (für p=1) z(G) = π + 1 (für p=1)
    - π = Anzahl der Verzweigungen über booleschen Bedingungen in G



McCabe-Maß: Programm-Graph





- z(G) = 4 (1.<2,3,7,8,10>, 2.<2,4,6,8,10>, 3.<2,4,6,8,9,8,10>, 4.<1.5>)
  <2,3,7,8,9,8,10> ist z.B. linear abhängig (= 3. – 2. + 1.)
- z(G) = e – n + 2p = 10 – 8 + 2 = 4 (10 Kanten, 8 Knoten, 1 Komponente)
- z(G) = π + 1 = 3 + 1 = 4 (Verzweigungen: Start, a, f)



Bewertung McCabe

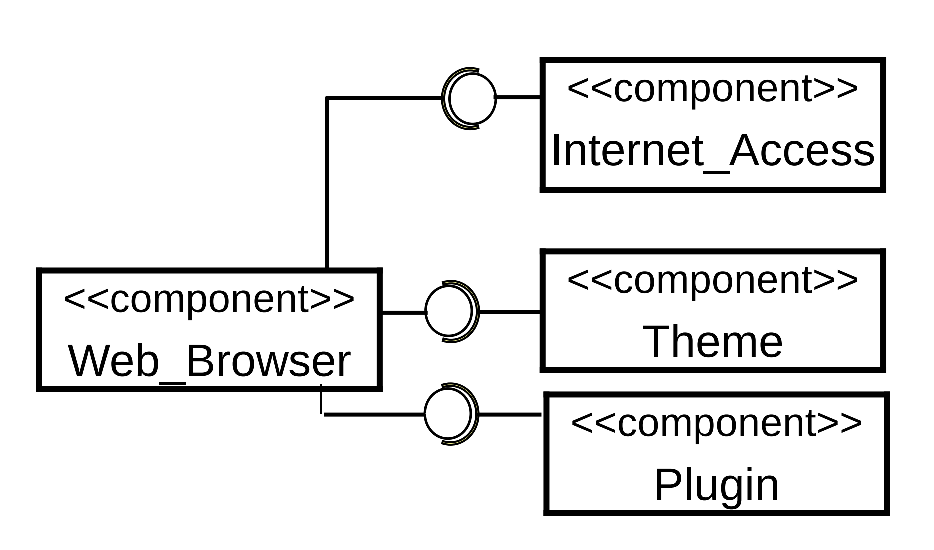
- Das McCabe Maß
  - ⇨ ist nur für Code anwendbar
  - ⇨ betrachtet nicht, wie kompliziert eine einzelne Anweisung ist, oder wie stark sie verschachtelt ist
  - ⇨ dennoch wird das McCabe Maß oft eingesetzt:
    "Zerlege jedes Programm mit z(G) ≥ 10 in Teilprogramme,
    so dass für jedes Teilprogramm gilt: z(G) < 10"

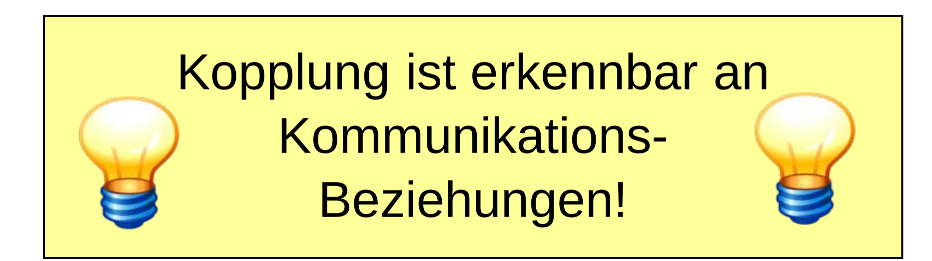


Aber was macht man, wenn es nicht um Code von Funktionen geht, sondern Kommunikation zw. Teilen??



Maße zur Beurteilung der der Kopplung zwischen Teilen des Systems: Informationsfluss-Analyse - IF4



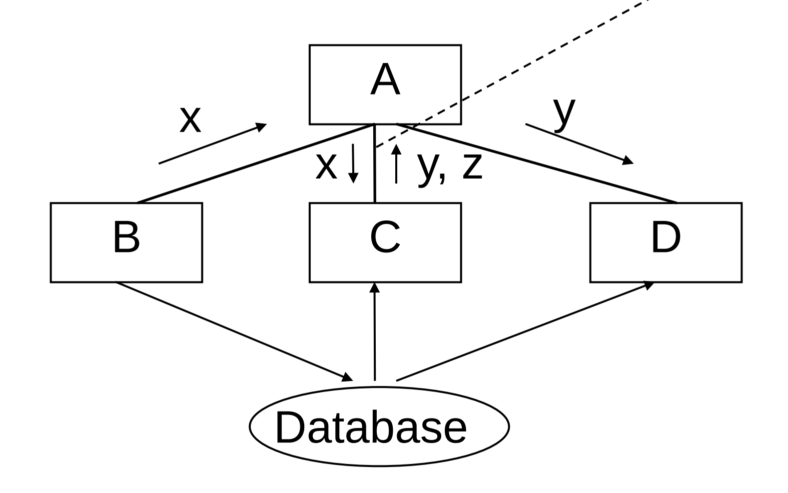


- "Informationsfluss-Analyse":
  - ⇨ FI_m: "fan-in" — Zahl der Datenflüsse, die zu einem Modul m führen
  - ⇨ FO_m: "fan-out" — Zahl der Datenflüsse, die von einem Modul wegführen
  - ⇨ $IF4_m(S) = (FI_m * FO_m)^2$ — Maßzahl für den Modul m im System S
  - ⇨ $IF4(S) = \sum_{i=1}^{n} (FI_i * FO_i)^2$ — Maßzahl für die Architektur des Systems S



Beispiel für die Informationsfluss-Analyse

- Beispiel:



A, B, C, D seien Funktionen und C ruft A auf mit Aufruf-Parametern y, z und Rück-gabeparameter x. *Oder* A, B, C, D seien Klassen und z.B. eine Funktion der Klasse A ruft eine Funktion der Klasse C mit Aufrufparameter x auf und eine Funktion der Klasse C ruft eine Funktion der Klasse A mit Aufrufparametern y, z auf, ...

Modul	fan-in (FI_i)	fan-out (FO_i)	$FI_i * FO_i$	$IF4_i = (FI_i * FO_i)^2$
A	**3**	**2**	**6**	**36**
B	**0**	**2**	**0**	**0**
C	**2**	**2**	**4**	**16**
D	**2**	**0**	**0**	**0**
				IF4(S) = 52



Relevanz der Maße für den Informationsfluss

- Empirische Studien haben gezeigt:
  - ⇨ Es besteht eine starke Abhängigkeit zwischen Informationsfluss-Maßen und dem Wartungsaufwand
  - ⇨ Beim "Altern" der Software entartet sehr oft die Struktur;
  - ⇨ IF4 und die verschiedenen IF4i können aufzeigen, ob und wo ein Re-Engineering notwendig bzw. sinnvoll wäre

- IF4 kann benutzt werden, um Architekturen miteinander zu vergleichen
  - ⇨ Deutlich geringeres IF4 ⇨ weniger starke Kopplung unter den Komponenten
  - ⇨ Ein Blob (mit Daten außerhalb des Blobs) zeigt ein sehr hohes IF4

- Welches ist die optimale Größe von Moduln?
  - ⇨ Zu diesem Zweck können kombinierte Maße wie (LOCi, IF4i) oder (McCabei, IF4i) benutzt werden, um ausgeartete Module zu lokalisieren



Messwerte und Schlussfolgerungen

Wer viel misst,
misst viel Mist!
(Reinhard K. Sprenger)

- Man kann fast alles messen...
  - ⇨ z.B. die Anzahl von Pizzadeckeln im Papierkorb - als Maß für den Terminverzug
- ...und aus den Messungen fast alles folgern!
  - ⇨ Die Auswahl der richtigen Metriken ist entscheidend
  - ⇨ Die Interpretation der Ergebnisse erfordert viel Erfahrung und Sachkenntnis
- Beispiel 1: Was sagt uns eine Mitarbeiter-Fluktuation von nur 2 Prozent?
  - ⇨ Dass die Angestellten zufrieden sind und es der Firma gut geht? oder
  - ⇨ dass die Mitarbeiter keine Alternative haben, weil die Firma in einer exotischen Nische lebt?
- Beispiel 2: Was sagt uns die Anzahl der vergangenen Tage seit Projektbeginn im Vergleich zu anderen Projekten?
  - ⇨ Nichts! Aber mit schicken Grafiken kann Wally seinen (unfähigen) Chef beeindrucken (Dilbert 1996)



Hochschule Darmstadt
Fachbereich Informatik

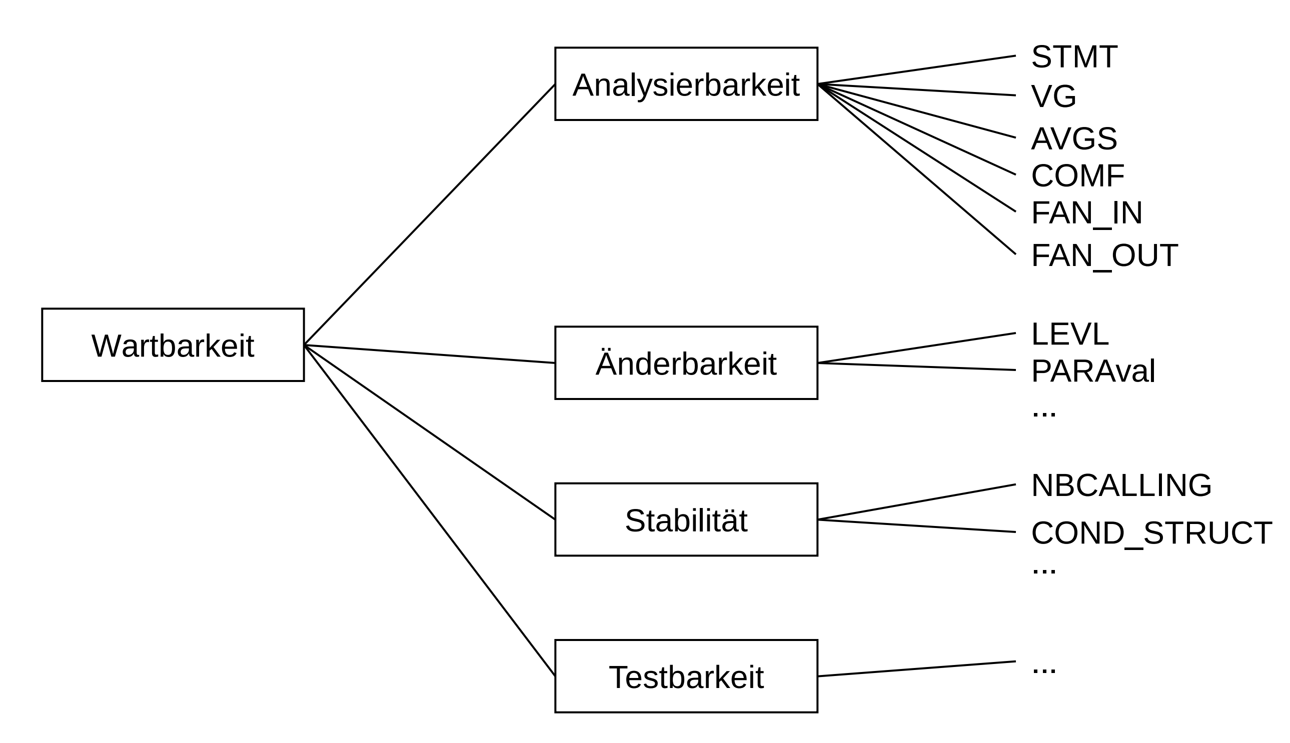
Software Engineering

10.1 Metriken und Tools



Tools

- Tools
  - ⇨ messen und berechnen grundlegende Metriken
  - ⇨ bieten komplexe zusammengesetzte Qualitätsmaße, die Schlussfolgerungen aus den gemessenen Werten ziehen (durch Verknüpfung und Gewichtung)



Teil des Qualitätsmodells des Testtools Logiscope der Fa. Verilog



Komplexe zusammengesetzte Qualitätsmaße

STMT	Anzahl der Statements einer Komponente Je größer die Anzahl der Statements in einem Programm, um so schwerer ist es zu verstehen, d.h. um so größer ist der Aufwand zur Fehlersuche oder bei Änderungen in der Wartung.
VG	zyklomatisches Maß nach McCabe
AVGS	durchschnittliche Länge der Statements Dieses Maß ist ein komplexes Maß und wird nach folgender Gleichung berechnet: AVGS=(N1+N2+1) / (STMT+1) wobei N1 = Gesamtanzahl der Operatoren (Halstead) N2 = Gesamtanzahl der Operanden Je länger eine Anweisung ist, um so größer ist der Aufwand sie zu verstehen.
COMF	Kommentarfrequenz Verhältnis von Kommentaren und Anweisungen.
FAN_IN	Zahl der Datenflüsse, die zu einem Modul führen Je größer diese Zahl für einen Modul ist, um so stärker wird der Modul durch seine Umgebung (die rufenden Module) beeinflusst.
FAN_OUT	Zahl der Datenflüsse, die von einem Modul wegführen Je größer diese Zahl für einen Modul ist, um so größer ist der Einfluss dieses Moduls auf seine Umgebung (⇨ besonders genau überprüfen)
⇨	Analysierbarkeit = f1*STMT + f2*VG + f3*AVGS + f4*COMF + f5*FAN_IN + f6*FAN_OUT



Darstellung der Messergebnisse: Kiviatdiagramm

Ausreißer!
⇨ Ursache untersuchen!

AXIS	LOW.	HIGH.	VALUE
COMF	0.20	+oo	***
AVGS	1.00	9.00	***
lc_stat	1.00	20.00	0.00
ct_vg	1.00	10.00	3.00
ct_bran	0.00	0.00	***
VOCF	1.00	4.00	***
dc_lvars	0.00	5.00	***
ic_param	0.00	5.00	***
dc_calls	0.00	5.00	0.00
ct_exit	0.00	1.00	1.00
ic_varpe	0.00	2.00	***
dc_calling	0.00	7.00	***
ct_path	1.00	60.00	4.00
LEVL	1.00	4.00	***

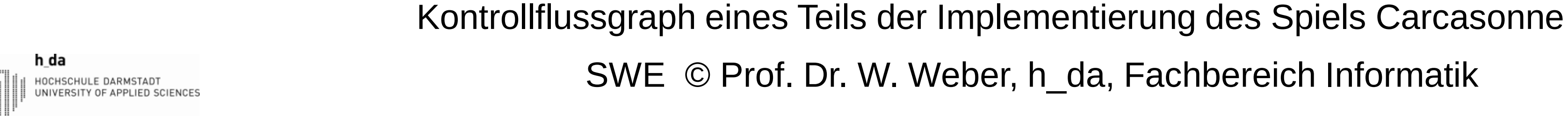


Darstellung der Messergebnisse: Kontrollflussgraph

Jede Verzweigung ist ein If- bzw. Case-Statement

Was bedeutet ein solches "Gebirge" für die Testbarkeit?

Kontrollflussgraph eines Teils der Implementierung des Spiels Carcasonne





Metriken und Tools

- Mit modernen Tools können sehr viele, verschiedene, vordefinierte Metriken auf die Arbeitsergebnisse (vor allem auf Code) angewandt werden
- Die Darstellungen erlauben eine schnelle Identifizierung von Ausreißern und die interaktive Analyse der Ursachen

  - ⇨ Die Versuchung ist groß, viele Metriken zu verwenden
  - ⇨ Die Versuchung ist groß, Ausreißer in allen Metriken zu analysieren
  - ⇨ Die intelligente Auswahl der Metriken und Grenzwerte wird umso wichtiger

Tools erleichtern den Umgang mit Metriken enorm –
aber sie ersetzen nicht die Denkarbeit!



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

10.2 Umgang mit Metriken



Gefahren von Metriken

- Metrik-Teufelskreis
  1. Das Projekt hat Terminverzug
  2. Das Management will den Grund für den Verzug verstehen und fordert detaillierte Metriken
  3. Die Mitarbeiter sammeln Daten für Metriken anstatt produktiv zu arbeiten
  4. GOTO1
- Metrik-Overkill
  - ⇨ ein von Metriken und Zahlen überzeugter Manager lässt zu viele Daten erfassen
  - ⇨ Die Mitarbeiter sammeln Daten für Metriken anstatt produktiv zu arbeiten

Dilbert: ... Here's my time report, in fifteen minute increments...



Umgang mit Metriken

- Metriken - ein normales und wichtiges Arbeitsmittel in modernen Projekten
  - ⇨ es sollten einige wenige angebrachte Metriken ausgewählt und konsequent verfolgt werden
  - ⇨ Metriken liefern Hinweise auf potenzielle Problemstellen – aber nicht jede Verletzung der Metrikvorgabe muss ein Problem sein (Manchmal sind Dinge auch einfach kompliziert)
  - ⇨ Mit modernen Tools können Metriken schnell berechnet und die Ergebnisse übersichtlich visualisiert werden. Dies liefert wertvolle Hinweise für die Entwicklung.
  - ⇨ Die Interpretation der Ergebnisse erfordert oft viel Erfahrung und Sachkenntnis
- Metriken müssen von allen Beteiligten mit Verstand eingesetzt werden
  - ⇨ Ein Entwickler kann Metriken immer austricksen! Kennt man die Berechnungsvorschrift, kann man die Metrik auch manipulieren.
  - ⇨ Ein Manager darf die Ergebnisse der Metriken nicht überbewerten

> "Nicht alles, was zählt, kann gezählt werden, und nicht alles was gezählt werden kann, zählt" (Albert Einstein)



Kontrollfragen Metriken

- Was halten Sie von LOC (Lines of Code) zur Beurteilung Ihres Arbeitsfortschritts? Welche Metrik ist besser?
- Wozu sind Metriken wichtig?
- Wofür können Metriken definiert werden?
- Warum sind Metriken oft bei Managern beliebt und bei Entwicklern unbeliebt?
- Was passiert, wenn zusätzliche Metriken / Maßzahlen erfasst werden, um den Verzug eines Projekts zu verstehen?
- Wie können Tools die Einhaltung von Metriken mit Sollwerten visualisieren?
- Welche Bedeutung hat das Maß IF4?

Können Sie jetzt in einer vernünftigen Weise mit Metriken umgehen?!



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

11. Vorgehens- und Prozessmodelle



Lernziel Vorgehensmodelle

- Sie sollen in diesen Kapitel lernen,
  - was ein Vorgehensmodell beinhaltet
  - wozu Vorgehensmodelle wichtig sind
  - was die verschiedenen Modelle auszeichnet
  - wodurch sich die verschiedenen Modelle unterscheiden

Anschließend können Sie zu einem Projekt ein Vorgehensmodell empfehlen!



Vorgehen in Projekten am Beispiel

	Projekt A	Projekt B
Projektart	x-te Weiterentwicklung mit langer Wartung	innovatives Produkt; basierend auf traditionellem Produkt
Beispiel	traditionelles Autoradio	neuartiges Navigationssystem
Auftraggeber	weiß genau was er will	weiß nicht wirklich was er will
Auftragnehmer	weiß genau, wie es geht	weiß auch nicht so genau was sinnvoll ist

- Wie würden Sie diese Projekte angehen?
- In welcher Reihenfolge würden Sie die bekannten Entwicklungsphasen durchlaufen?
- Würden Sie die Phasen einmal vollständig, oder eher inkrementell durchlaufen?



Die Grundidee für Vorgehensmodelle am Beispiel

- Die Projekte erfordern ein unterschiedliches Durchlaufen der Phasen!

- ähnliche Projekte kann man immer wieder so durchführen
- die Abarbeitung der Phasen kann als Modell dokumentiert werden



Warum braucht man Vorgehensmodelle?

- Softwareentwicklung ist ein komplexer Prozess und erfordert ein systematisches, geplantes Vorgehen

- Ein Vorgehensmodell
  - ⇨ gibt einen bewährten Leitfaden für den Entwicklungsprozess
  - ⇨ erlaubt die Beschreibung, Optimierung und Zertifizierung von Entwicklungsprozessen

⇨ Vorgehensmodelle unterstützen die geplante, erfolgreiche Durchführung von Softwareentwicklungen, d.h. innerhalb definierter Zeit, Kosten und Qualität!



Was ist ein Vorgehensmodell?

- Ein Vorgehensmodell legt fest:
  - ⇨ die Reihenfolge des Arbeitsablaufs (Entwicklungsstufen, Phasenkonzepte)
  - ⇨ Jeweils durchzuführende Aktivitäten in den Phasen
  - ⇨ Definition der Teilprodukte (Arbeitsergebnisse)
  - ⇨ Fertigstellungskriterien und Abnahmekriterien
  - ⇨ Notwendige Mitarbeiterqualifikationen
  - ⇨ Verantwortlichkeiten und Kompetenzen
  - ⇨ Anzuwendende Standards, Richtlinien, Methoden und Werkzeuge
- Für ein konkretes Softwareentwicklungsprojekt muss das Vorgehensmodell konkretisiert werden (neu-deutsch: Tailoring)

> ⇨ Ein Vorgehensmodell ist eine systematisch dokumentierte Vorlage für die Durchführung und das Management von Projekten!



Welche Vorgehensmodelle gibt es?

Klassische Modelle

- (Code & Fix)
- Wasserfall-Modell
- Evolutionäres Modell
- Inkrementelles Modell
- V-Modell
- Prototypen-Modell
- ...

↑

Unterscheiden sich in der Abfolge von Phasen

neue, komlexe Modelle

- Rational Unified Process (RUP)
- V-Modell XT
- Agile Entwicklung
- ...

↑

Definieren den Gesamtprozess mit Definition der Aktivitäten, Arbeitsergebnissen etc. der einzelnen Prozessschritte (Phasen)



Code & Fix

Erstelle erste Version
Ausmusterung
Ändern bis der Kunde zufrieden ist
Einsatz der Version
Ein häufig anzutreffendes "Vorgehensmodell"
Entwicklung
Verbesserung
Nicht jedes Vorgehens-modell sorgt für gute Qualität!


h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Bewertung von Code & Fix

Vorteile

- Geringer Managementaufwand
- Zufriedene Kunden

Nachteile

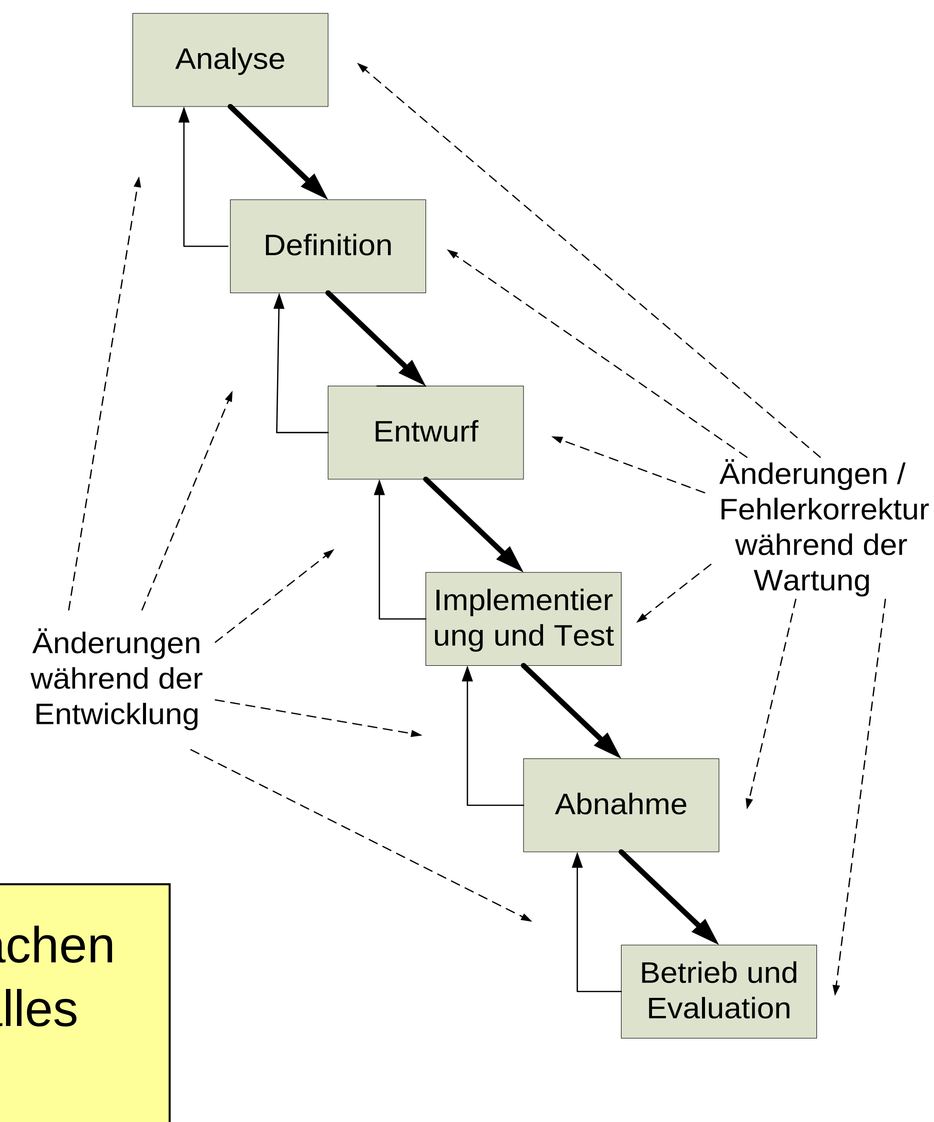
- Schlechtere Wartbarkeit, Zuverlässigkeit und Übersichtlichkeit des Codes
- Starke Abhängigkeit vom individuellen Programmierer
- Differenzen über Funktionsumfang zwischen Entwickler und Anwender
- Keine Entwicklung von Dokumentation und Testfällen

⇨ Für große Projekte ungeeignet!



Das Wasserfall-Modell

- Sequentieller Entwicklungsablauf
- Top-Down-Vorgehen
- Durchführung jeder Aktivität in der richtigen Reihenfolge und in vollem Umfang
- Abgeschlossene Dokumentation am Ende jeder Aktivität
- Iterationen nur zwischen zwei aufeinanderfolgenden Stufen





"Wir machen gleich alles richtig!"



Bewertung des Wasserfall-Modells

Vorteile

- Extrem einfaches Modell
- Geringer Management-Aufwand
- Disziplinierter, kontrollierbarer und sichtbarer Prozessablauf

Trotz aller Nachteile sehr beliebt bei Managern
- und immer noch weit verbreitet!

Auch das traditionelle V-Modell ist ein Wasserfallmodell

Nachteile

- Strenge Sequentialität oft nicht sinnvoll/machbar
- Möglichkeit von Feedback kaum gegeben
- Erkennen von Problemen erst am Ende
- Benutzerbeteiligung nur bei Anforderungen und im Betrieb
- Gefahr einer zu starken Gewichtung der Dokumentation
- Lange Projektlaufzeit, keine Zwischenversionen zu früheren Zeitpunkten

Das evolutionäre Modell

- Kernanforderungen des Auftraggebers führen direkt zur „Nullversion" (wird ausgeliefert)
- Konzentration auf lauffähige Teilprodukte
- Stufenweise Entwicklung durch Modell, Steuerung durch die Erfahrung der Benutzer
- Vermeidung der Implementierung eines Produkts gleich in voller Breite
- Neue Anforderungen ⇨ neue Version
- Integration von Pflegeaktivitäten in den Prozess

Baue erst den Systemkern – dann ergänze den Rest!



Bewertung des evolutionären Modells

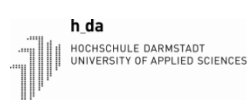
Vorteile

- Einsatzfähige Produkte für den Auftraggeber in kurzen Abständen
- Integration der Erfahrungen der Anwender in die Entwicklung
- Überschaubare Projektgröße
- Korrigierbare Entwicklungsrichtung
- Keine Ausrichtung auf einen einzigen Endtermin

Nachteile

- Eventuell mangelnde Flexibilität der Nullversion zur Anpassung an unvorhersehbare Evolutionspfade
- Eventuell komplette Änderung der Architektur in späteren Versionen

Gut geeignet, wenn der Auftraggeber noch nicht sicher ist, was er will!



Das iterativ-inkrementelle Modell

- Möglichst vollständige Erfassung und Modellierung der Anforderungen an das zu entwickelnde Gesamtprodukt
- Realisierung der nächsten Ausbaustufe unter Berücksichtigung der Erfahrungen des Auftraggebers mit der laufenden Version
- Einsatzfähiges System für den Auftraggeber schon nach relativ kurzer Zeit

Analyse des Gesamtsystems

Def. des Gesamtsystems inkl. Zyklenplanung

Entwurf des Teilsystems

Implementierung und Test des Teilsystems

Abnahme des Teilsystems

Betrieb & Evaluation des Teilsystems

Anpassungen auf Grund von Erfahrung mit dem Teilsystem

zum nächsten Zyklus

Änderungen

Plane erst das Gesamtsystem – dann baue in Zyklen die Teilsysteme!



Bewertung des inkrementellen Modells

Vorteile

- zusätzlich zu den Vorteilen der evolutionären Entwicklung:
- Inkrementelle Erweiterungen passen zum bisherigen System

Nachteile

- Vollständige Spezifikation nicht immer möglich
- die Entwicklung der ersten Version dauert länger als beim evolutionären Modell

Gut geeignet, wenn der wesentliche Inhalt klar ist,
aber viele Details offen sind!

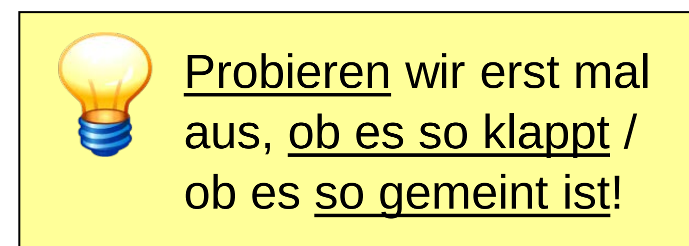


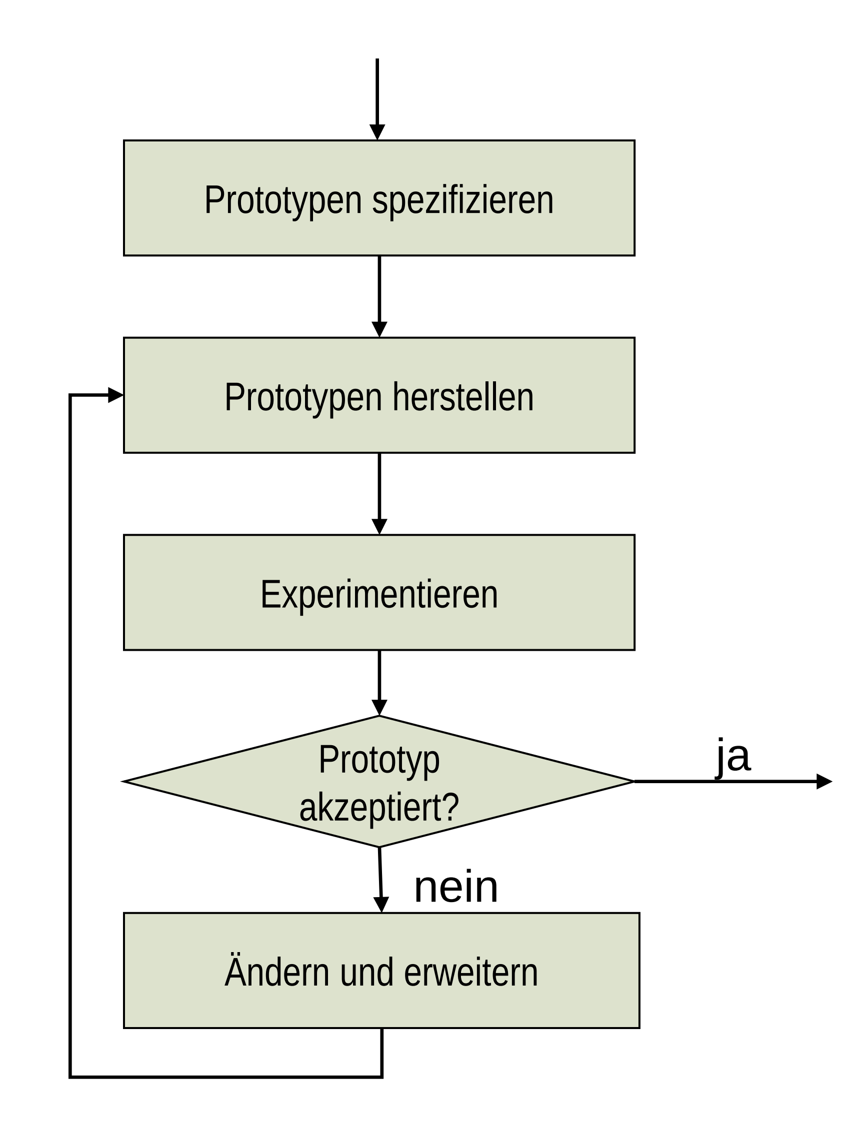
Prototypen

- Erstellung von lauffähigen Prototypen zur frühzeitigen Klärung von Problemen
  - ⇨ Auswahl alternativer Lösungsmöglichkeiten
  - ⇨ Sicherstellung der Realisierbarkeit
  - ⇨ Vervollständigen, Korrigieren von Anforderungen
- Einbeziehen von Anwendern in die Entwicklung

Prototypen(-Phasen)-Modell

- Schrittweise Weiterentwicklung des ersten Prototyps / der frühen Produktversionen (add-on-Prototyp s. u.)





Prototypen-Modell



Prototypen

- Verwendung von Prototypen nicht nur im Prototypen-Modell, sondern auch bei Verwendung anderer Vorgehensmodelle zur Prüfung der Machbarkeit, Verbesserung von Anforderungsspezifikation und Entwurf.

Definition:

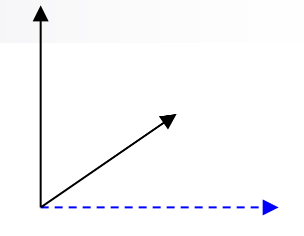
- Ein **Prototyp** ist ein Experimentiersystem, mit dessen Hilfe Fragestellungen zu Eigenschaften des endgültigen Produkts oder seiner Einsatzumgebung geklärt werden
- **Prototyping** ist die systematische Anwendung von Prototypen
- 3 Klassifikationskriterien:

3.
2.
1.

1. Was wird durch den Prototyp getestet?
2. Wie ist der Prototyp aufgebaut?
3. Inwieweit findet der Prototyp Verwendung bei der Implementierung des endgültigen Systems?



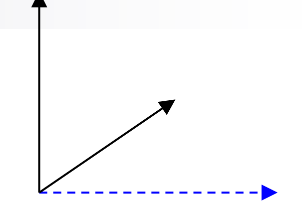
1. Was wird durch den Prototyp getestet? (I)



- Benutzer-Prototyp (Explorativer Prototyp)
  - ⇨ dient als Kommunikationsgrundlage zwischen Endbenutzer und den Systementwicklern
  - ⇨ Spätere Arbeitsweise wird durchgespielt
  - ⇨ Schwächen und Fehler der Spezifikation werden erkannt und vor der Weiterarbeit beseitigt
  - ⇨ Beispiel:
    - Implementierung (eines Teils) der Benutzeroberfläche
      - ohne oder
      - mit darunterliegender Logik und Datenbank

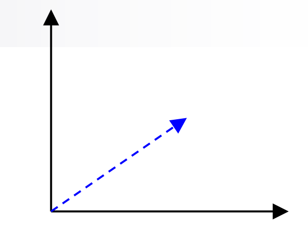


1. Was wird durch den Prototyp getestet? (II)



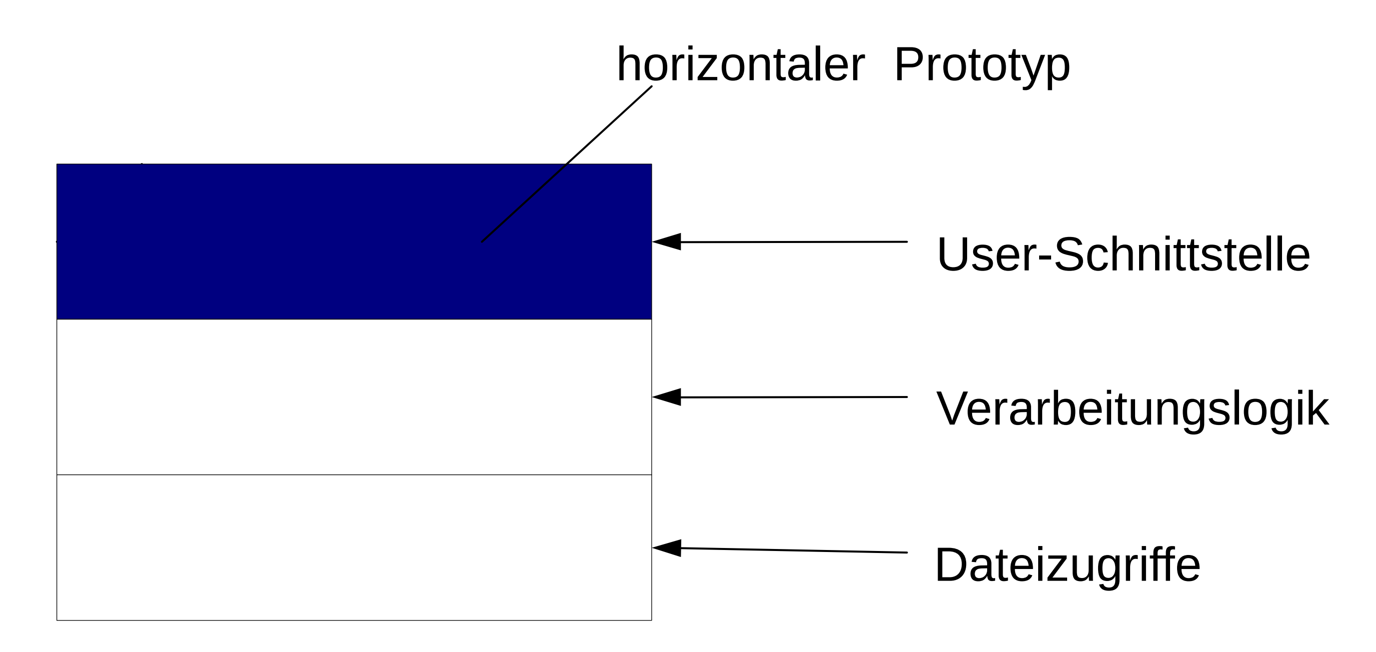
- Technischer Prototyp (Experimenteller Prototyp, Labormodell)
  - ⇨ dient den Entwicklern intern als Bewertungsgegenstand, der Fragen der technischen Umsetzung und Realisierbarkeit klären soll
  - ⇨ Beispiel für Zielsetzungen von technischen Prototypen:
    - Überprüfung der Machbarkeit
    - Abschätzung des Aufwandes / des Zeitbedarfs / der Kosten der Realisierung
    - Vergleich von Alternativen
    - Performance-Tests
    - Fragen der Konfiguration (Anschluss von Druckern, Vernetzung, …)
    - Einsatz von Tools
    - ......

2. Wie ist ein Prototyp aufgebaut? (I)



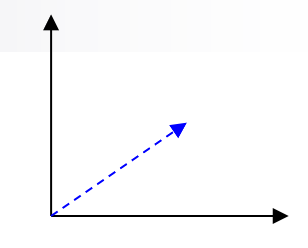
horizontaler Prototyp

- z. B. nur Schnittstelle zum Benutzer
- Mit Hilfe solch eines Prototyps kann das Arbeiten mit dem System gezeigt werden
- Da die tieferen Ebenen fehlen, liefert das System normalerweise falsche Werte



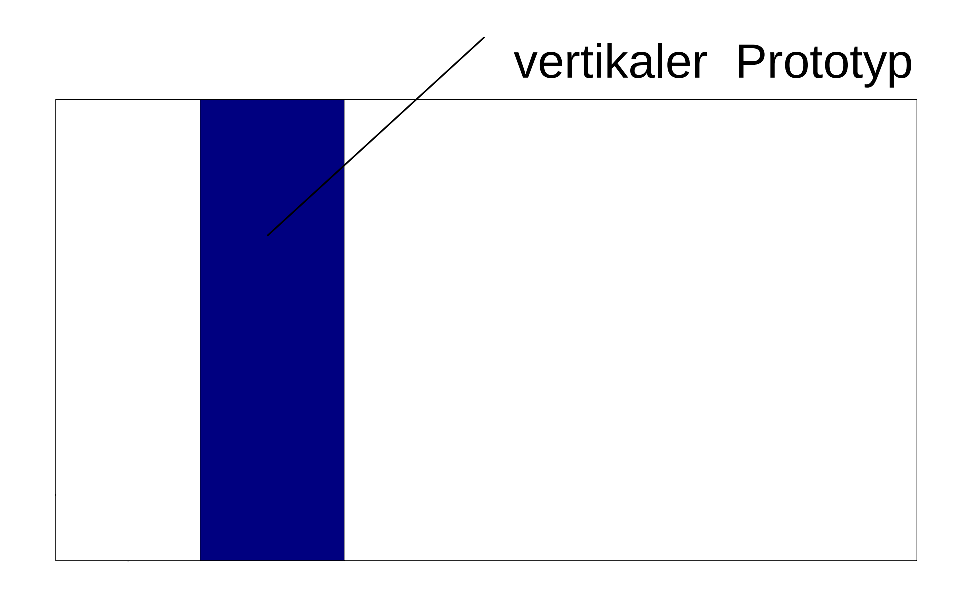


2. Wie ist ein Prototyp aufgebaut? (II)



vertikaler Prototyp

- exemplarisch ausgewählte Teilfunktionen des zukünftigen Systems vollkommen realisieren
- z.B. Performanz einer kritischen Anwendung
- auch als explorativer Prototyp

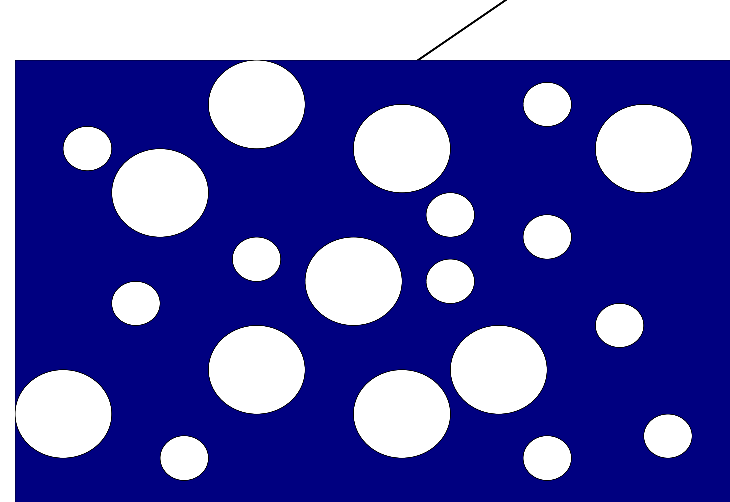


2. Wie ist ein Prototyp aufgebaut? (III)

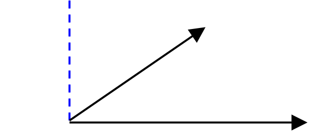
Gesamtsystem mit Lücken

⇨ alle Teilsysteme realisiert außer z.B.

- Plausibilitätsprüfungen
- Ausnahme- und Fehlersituationen
- komplexe Berechnungsroutinen
- ....



3. Inwieweit findet der Prototyp Anwendung im endgültigen System?



Throw-away-Prototype (Quick-and-dirty-P, Rapid-Specification-P.)

- ⇨ findet keine Verwendung bei nachfolgender Implementierung
- ⇨ dient einzig und allein zur Entwicklung korrekter, robuster Spezifikationen

Add-on-Prototype (Rapid-Cyclic-Prototype)

- ⇨ wird bei der Implementierung des Zielsystem mitverwendet
- ⇨ ausgehend von einem ersten Add-on-Prototype wird zum Zielsystem weiterentwickelt

z. B. Prototypenmodell:

In jeder Iterationsstufe

- ⇨ wird der schon entwickelte Teil weiter verbessert
- ⇨ wird System vom Umfang her erweitert
- ⇨ wird weiterentwickelt, bis das Zielsystem vorliegt



Bewertung Prototypen / Prototypen-Modell

Vorteile

- Sinnvolle Integration in andere Vorgehensmodelle möglich
- Schaffung einer starken Rückkopplung zwischen Endbenutzern und Herstellern
- Schnelle Erstellung von Prototypen durch geeignete Werkzeuge

Nachteile

- Erhöhter Entwicklungsaufwand durch zusätzliche Herstellung von Prototypen
- Gefahr der Umwandlung eines „Wegwerf-Prototyps“ zu einem Teil des Endprodukts aus Termingründen
- Prototypen "ersetzen" fehlende Dokumentation

Prototypen sind geeignet, um das Entwicklungsrisiko zu reduzieren!



Rational Unified Process



- Der Rational Unified Process (RUP) ist ein von der Firma Rational Software parallel zur Unified Modeling Language entwickeltes Vorgehensmodell zur Softwareentwicklung
- Der RUP definiert 6+3 grundlegende ***Disziplinen***

  und
- orthogonal dazu 4 ***Phasen***



Disziplinen (Arbeitsschritte) im RUP

Disciplines (Arbeitsschritte)

Geschäftsprozessmodellierung

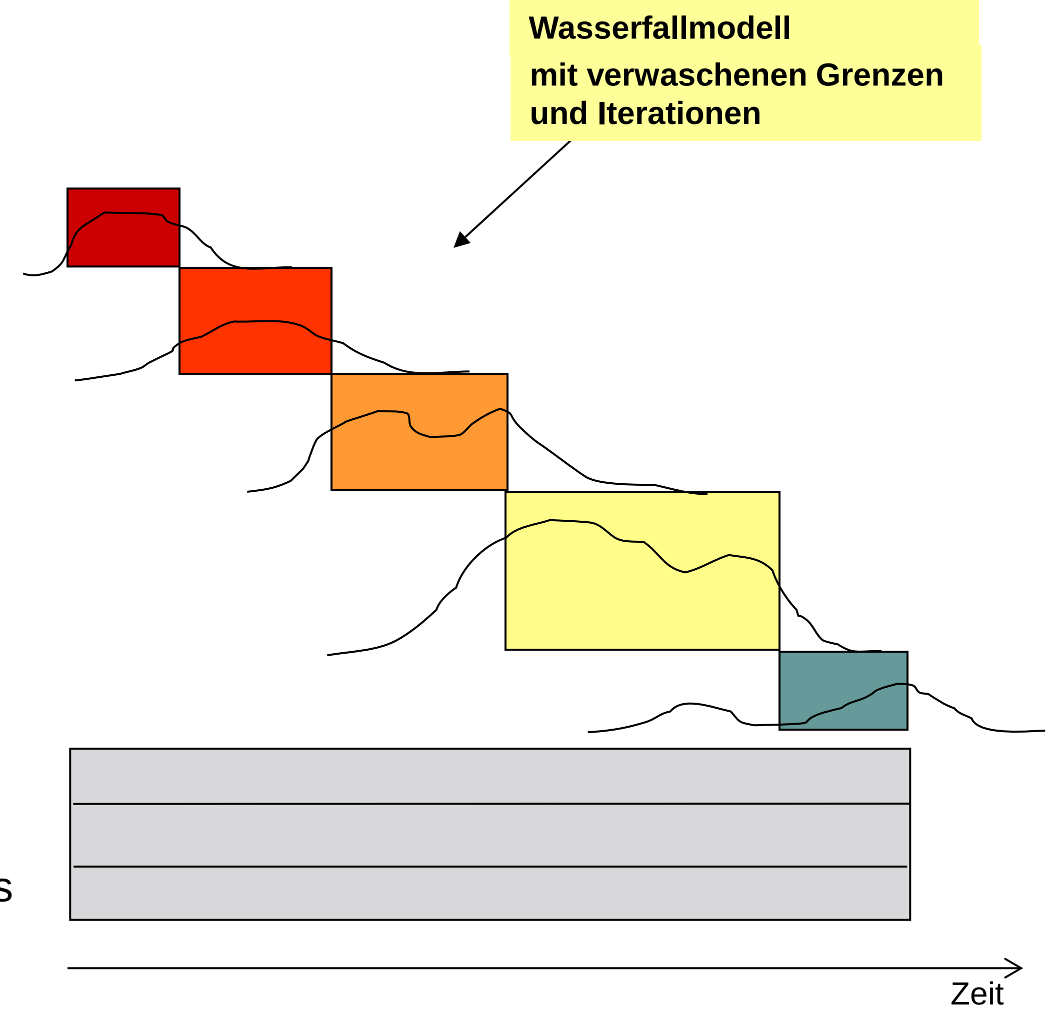
Anforderungsanalyse

Analyse & Design

Implementierung
Test

Auslieferung

Projektmanagement
Konfigurations- und Änderungsman.
Erst. d. Umgebung, Konfig. d. Prozesses





Disziplinen (Arbeitsschritte) und Phasen im RUP

Phases

Disciplines (Arbeitsschritte)



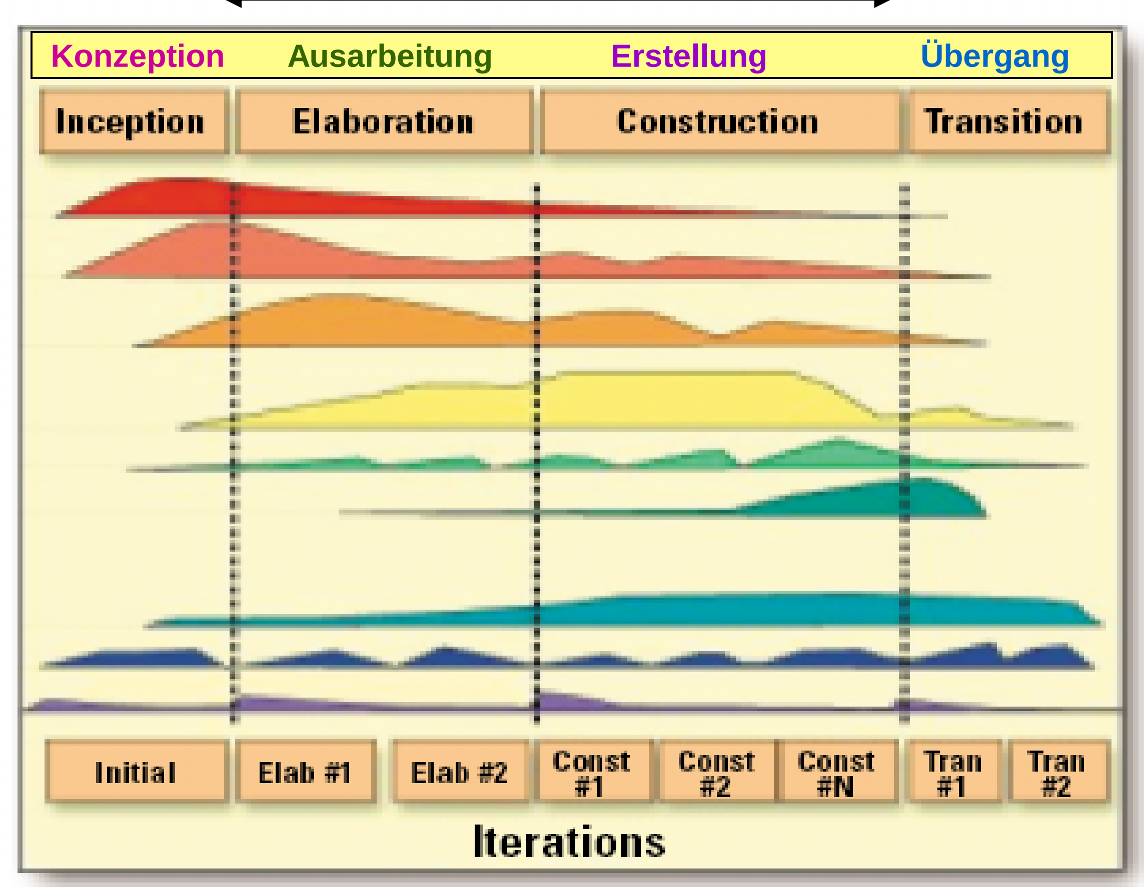
Organization along content

Core Process Workflows

- Business Modeling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment

Core Supporting Workflows

- Configuration & Change Mgmt.
- Project Management
- Environment



Quelle: Rational Unified Process 2000 Tool



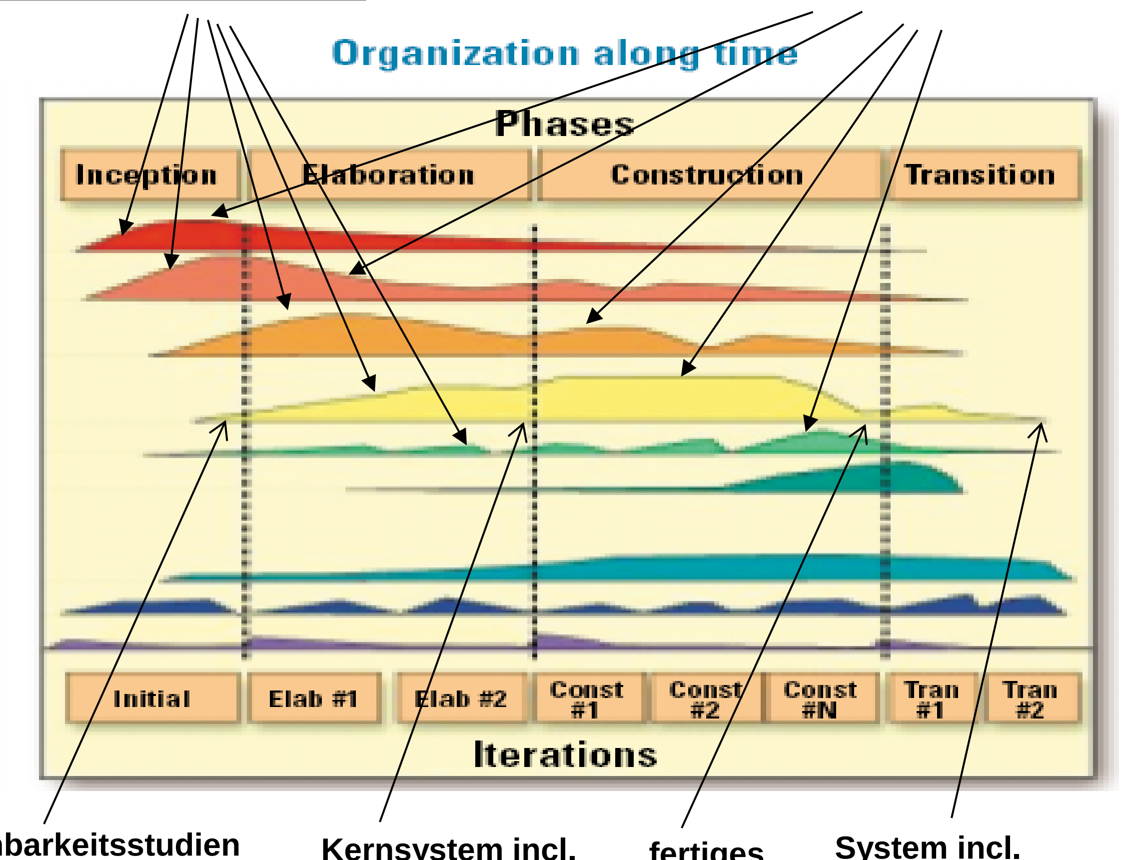
Disziplinen (Arbeitsschritte) und Phasen im RUP

Beispiel:

Auftragserfassung im Kernsystem

Auftragserfassung im fertigen System (mit Sonderfällen)

Disciplines (Arbeitsschritte)



Implementierungs-Meilensteine:

Machbarkeitsstudien mit Prototyping

Kernsystem incl. Tests

fertiges System

System incl. Anpassungen



Bewertung: RUP / UP

Vorteile

- Durch die Überlappung verkürzt sich Dauer des Projektes
- Unterstützt die Planung und Steuerung von Projekten durch Standardisierung

  Für jeden (Teil-)Prozess:

  Welche Arbeiten,

  welche Arbeitsergebnisse,

  wer verantwortlich.
- Anpassbar an projektspezifische Anforderungen (Tailoring)
- Gute Anbindung zur UML
- Gut dokumentiertes Modell

Nachteile

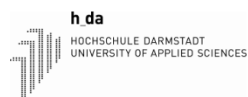
- RUP stellt hohe Anforderungen an das Management durch die hohe Parallelität des Entwicklungsprozesses
- Die Anforderungen werden relativ früh festgelegt

RUP ist ein gut dokumentiertes und modernes iteratives Modell

– aber nicht evolutionär!



Rest von RUP nur zur Info, nicht prüfungsrelevant



Unterschied zwischen Rational Unified Process (RUP) und Unified Process (UP)

- UP wurde 1999 von Jacobson veröffentlicht
- RUP ist eine konkrete Implementierung des UP durch die Firma Rational nach 1999
- RUP wird von Rational (inzwischen aufgekauft von IBM) als Produkt vertrieben, Rational bietet für RUP eine Reihe von darauf abgestimmte Formularen und Werkzeugen an.
- UP teilt Projekte in folgende Disziplinen (Arbeitsschritte) ein:
  - ⇨ Anforderungen
  - ⇨ Analyse
  - ⇨ Entwurf
  - ⇨ Implementierung
  - ⇨ Test

  Wir sehen, es fehlen gegenüber dem RUP:
  - ⇨ Geschäftsmodellierung
  - ⇨ Auslieferung
  - ⇨ Konfigurations- und Änderungsmanagement (siehe Kap 9)
  - ⇨ Projektmanagement (mit Risiko-, Personal-, Budget- und Vertragsmanagement)
  - ⇨ Erstellung und Pflege der Infrastruktur

  Selbstverständlich brauchen wir in UP auch die in RUP explizit zusätzlich erwähnten Disziplinen.



Phasen und Ergebnisse

Siehe auch S. 39-45 in [Arlow, Neustadt]

Ergebnisse (Produkte) der	Inception Phase ("Konzeption", "Proj.definition")	Elabor.
Geschäftsprozess-Modellierung	Haupt-Geschäftsfälle sind definiert	.
Anforderungsdefinition	Projektziele, Schlüsselanforderungen (Anwendungsfälle Kernsystem), Akteure Visionen-Dokument mit Hauptanforderungen, Funktionen und Grenzen des Gesamtsystems , Priorisierung der Anforderungen, Projekt-Glossar,	.
Analyse, Design	Erster Entwurf der Systemarchitektur, Schnittstellen zu anderen Systemen Nachweis der Machbarkeit und Brauchbarkeit durch technische Studien	.
Implementierung u. Test	Nachweis der Machbarkeit und Brauchbarkeit durch definierte technische / explorative Prototypen	.
Auslieferung	.	.
ProjektManagement	Kosten-, Zeitschätzung und Projektplanung mit genau definierten Meilensteinen, Risikoanalyse, Nutzen des Systems ist dargestellt, Kosten-Nutzenanalyse liegt vor, Groben Kosten- und Terminplan liegt vor.	.
Env: Infrastruktur eingerichtet	Stakeholder haben den obigen Punkten zugestimmt und entschließen sich, das Projekt fortzuführen	



Phasen und Ergebnisse

Siehe auch S. 39-45 in [Arlow, Neustadt]

Ergebnisse (Produkte) der	Elaboration Phase ("Ausarbeitung ", "Entwurf")	Constr.
Geschäftsprozess-Modellierung	Mit den Stakeholder überarbeitetes endgültiges Gesamtgeschäftsprozessmodell als Workflow mit verbalen Beschreibungen liegt vor	.
Anforderungsdefinition	UML Use Case Diagramm (fertig), UML Use Case Beschreibungen (fast fertig) nichtfunktionale Anforderungen sind definiert	.
Analyse, Design	Statische und dynamische UML-Modelle zu Analyse (fertig), Architektur (fertig), **Design (zum großen Teil fertig)** Benutzerdokumentation (begonnen)	.
Implementierung u. Test	Ein belastbares, robustes, ausgetestetes, **ausführbares Kernsystem** (Prototyp) Testplan und Testfälle (fast fertig)	.
Auslieferung	Kernsystem (zum Prüfen) ausgeliefert	.
ProjektManagement	Detaillierter Projektplan, mit einer realistischen Einschätzung über benötigte Zeit, Geld und weitere Resourcen, verfeinerte Risikoanalyse, Kosten-Nutzenanalyse liegt vor.	.

Geschäftsmodell ist mit Projektplan abgestimmt und von Stakeholdern genehmigt
Stakeholder entschließen sich, das Projekt fortzuführen



Phasen und Ergebnisse

Siehe auch S. 39-45 in [Arlow, Neustadt]

Ergebnisse (Produkte) der	Construction Phase ("Erstellung", "Konstruktion")	Trans.
Geschäftsprozess-Modellierung	Alle Produkte sind auf dem neuestem Stand und konsistent zueinander (fertig)	.
Anforderungsdefinition	Anforderungsdokument mit zusätzlich gefundene Anforderungen (fertig)	.
Analyse, Design	**Benutzerdokumentation und Help-Bildschirme fertig** Alle übrigen Teile sind fertig	.
Implementierung u. Test	Ein genügend belastbares, robustes, ausgetestetes, ausführbares **Gesamtsystem**, um es an den Benutzer auszuliefern, **Beta-Test fertig**	.
Auslieferung	**Teilsysteme wurden bereits ausgeliefert** Schulungsunterlagen sind erstellt	.
ProjektManagement	Endgültiger Installations- und Einführungsplan Nachkalkulation der Kosten und Schwachstellenfindung als Grundlage für nächste Projekte	.
Env: Zielumgebung vorbereitet		

Stakeholder habe der Auslieferung der Software zugestimmt

Phasen und Ergebnisse

Siehe auch S. 39-45 in [Arlow, Neustadt]

Ergebnisse (Produkte) der	Transition Phase ("Übergang", "Übergabe")
Geschäftsprozess-Modellierung	Alle Produkte sind auf dem neuestem Stand und konsistent zueinander
Anforderungsdefinition	Alle Produkte sind auf dem neuestem Stand und konsistent zueinander
Analyse, Design	Alle Produkte sind auf dem neuestem Stand und konsistent zueinander
Implementierung u. Test	Installiertes abgenommenes System, Benutzer-Test ist abgeschlossen, gefundene Fehler sind korrigiert, SW ist an evtl. zusätzlich gefundene / geänderte Anforderungen angepasst
Auslieferung	**Alle für den Kunden notwendigen Produkte sind ausgeliefert**, Software ist installiert u. für spezielle Benutzer konfiguriert, Daten sind übernommen, Schulungen sind durchgeführt, Benutzer wurde bei der Handhabung des Systems unterstützt, Projekt-Abschluss Review ist durchgeführt.
ProjektManagement Env: Zielsystem eingerichtet	Sämtliche Projektdaten sind im Unternehmen archiviert Plan zur weiteren Unterstützung des Benutzers ist abgestimmt

Stakeholder sind (hoffentlich) zufrieden



Phasen und Ergebnisse

- Iterationen
  - ⇨ Bei größeren Projekten sind insbesondere die Phasen Ausarbeitung, Erstellung und Übergang in mehrere Iterationen unterteilt.
  - ⇨ Nach jeder Iteration ist man den Endziel etwas näher gekommen und es sind weitere Ergebnisse (Produkte) entstanden.
  - ⇨ Es entstehen u. a. Zwischenversionen des lauffähigen Programms (incl. Dokumentation), die entweder nur intern zur Bewertung des Arbeitsfortschritts dienen oder die an den Kunden zum Testen oder schon für den produktiven Einsatz ausgeliefert werden.
  - ⇨ Man spricht von einem iterativ inkrementellen Prozess, da in jeder Iteration ein Inkrement (Produkte eines Arbeitsschritts, d. h. UML-Modelle, ..., lauffähiger Code) zum System hinzugefügt wird. (Allerdings können die verschiedenen Arbeitsschritte zu einem Programmteil in unterschiedlichen Iterationen ablaufen! – s. nächste Folie)
  - ⇨ Im Gegensatz zu dem normale íterativ-inkrementellen Ansatz können sich hier die Zyklen überlappen, d. h. der Folgezyklus beginnt, bevor der Vorgängerzyklus abgeschlossen ist.
  - ⇨ Nicht nur die Ziele und Produkte der einzelnen Disziplinen,

    sondern auch die Ziele und Produkte der Iterationen (Meilensteine) müssen definiert sein und durch Reviews überprüft werden.



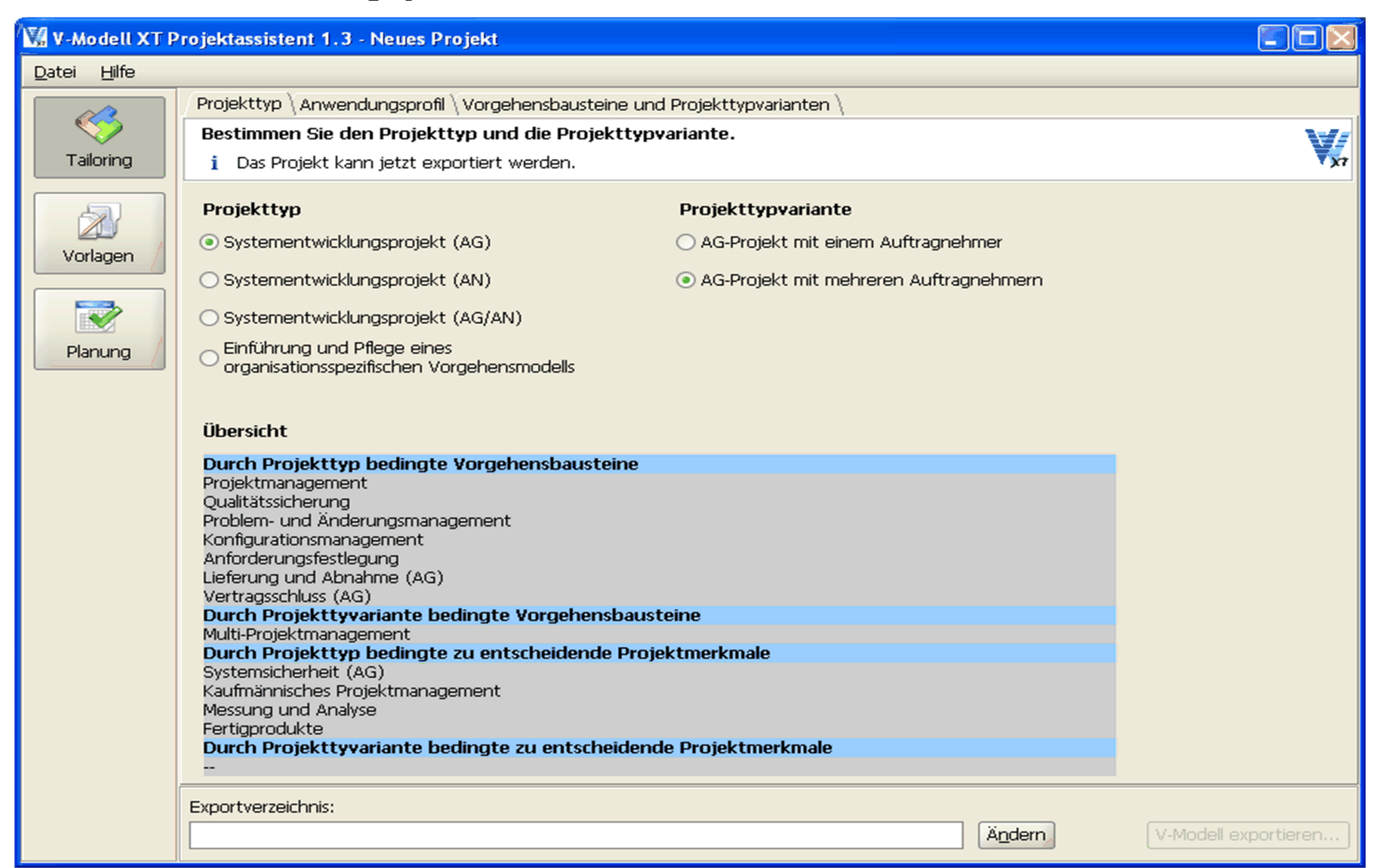
V-Modell-XT

- Verbindliches Vorgehensmodell bei der Vergabe von Entwicklungsprojekten durch Bundesbehörden
- V-Modell (1988) -> V-Modell 97 (1997) -> V-Modell XT (2005)
- XT = extreme Tailorig -> kann durch Tailoring an unterschiedliche Projektarten angepasst werden
- Es wird vom Staat ein rechnerunterstütztes Tailoringsystem angeboten
- Die zugeschnittene Ausprägung des V-Modell-XT ist abhängig von
  - ⇨ dem „Projekttyp“ (Projekt aus Auftragnehmer oder Auftraggeber-Sicht)
  - ⇨ weiteren „Projektmerkmalen“ (Fertigprodukt, Projekt mit hohem Risiko,...)
  - ⇨ der „Projektdurchführungsstrategie“ (inkrementelle Entwicklung, komponentenbasierte Entwicklung, agile Entwicklung, Wartung, ...)
  - ⇨ ausgewählten „Entscheidungspunkten“ (=Meilensteine: z. B. System spezifiziert, ..) und einer vorgeschlagenen Reihenfolge der Entscheidungspunkte (Workflow)
  - ⇨ Zu jeden Entscheidungspunkt werden Produkte vorgeschlagen, die beim Eintreten des Entscheidungspunkts vorliegen müssen
  - ⇨ gewissen Projekttypen können gewissen „Vorgehensbausteine“ zugeordnet werden (Vertragsabschluss(AG), Evaluierung Fertigprodukte, Systemerstellung)

Alle Abbildungen in diesem Kapitel aus : V-Modell XT, Release 1.3



V-Modell-XT - Tailoringsystem



V-Modell-XT - Entscheidungspunkte, Projektdurchführungsstrategie

Auftraggeber

Entscheidungspunkte

1 Projekt genehmigt
2 Projekt definiert
3 Anforderungen festgelegt
4 Projekt ausgeschrieben
5 Projekt beauftragt
6 Abnahme erfolgt
8 Projekt abgeschlossen
7 Änderungsplan festgelegt

Ausschreibung
Angebot
Vertrag

1 Projekt genehmigt
2 Angebot abgegeben
3 Projekt beauftragt
4 Projekt definiert
5 System spezifiziert
6 System entworfen
7 Feinentwurf abgeschlossen
8 Systemelemente realisiert
9 System integriert
10 Lieferung durchgeführt
11 Abnahme erfolgt
12 Änderungsplan festgelegt
13 Projekt abgeschlossen

Auftragnehmer

jeder Entscheidungspunkt
hat mehrerer Produkte



Rest von V-Modell-XT ob nur zur Info, nicht prüfungsrelevant



V-Modell-XT - Metamodell

Projektmerkmale

1..3

1..2

Projekttyp

1..*

optional

Vorgehensbausteingruppe

1

1..4

1..*

mandatory

1..*

1

1..9

Projektdurchführungsstrategie

1..*

Vorgehensbaustein

Produkt-Gruppe

{ordered as workflow}

1..*

1..*

1

1..*

1..*

1..*

1..*

1

Entscheidungspunkte

1..*

1..*

Produkte

is dependant on

1..*

contribute

1..*

responsible

1..*

1

1

1..*

1

Rolle

Aktivität

1..*

1..*

Aktivitäts-Gruppe

1

1..*

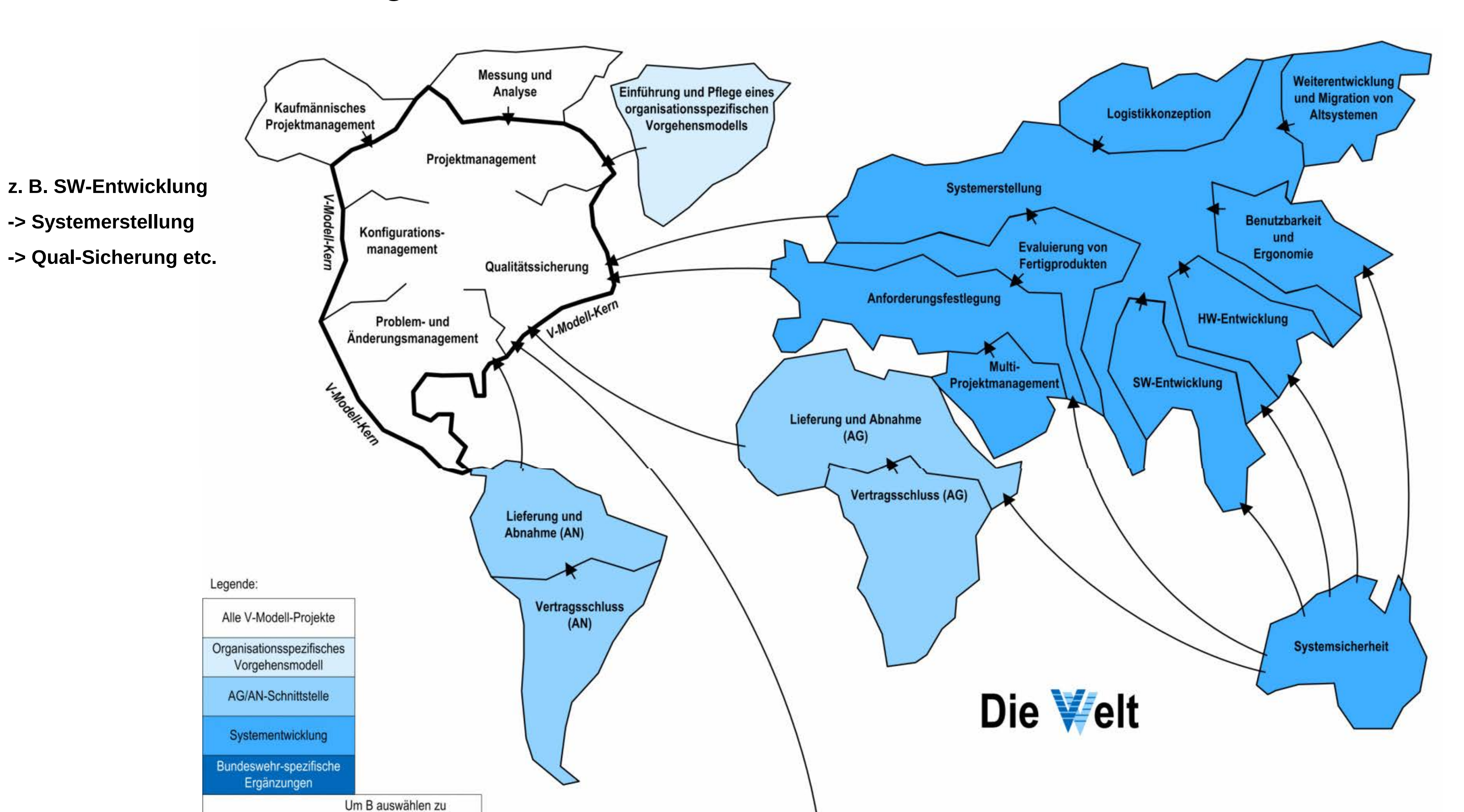
Aktivitätsschritt

V-Modell-XT - Vorgehensbausteine

z. B. SW-Entwicklung

-> Systemerstellung

-> Qual-Sicherung etc.



V-Modell-XT - Vorgehensbausteine, Entscheidungspunkte

	Projekttyp	Projektdurchführungsstrategien	
	Verpflichtende Vorgehensbausteine	Optionale Vorgehensbausteine	Entscheidungspunkte
	Systementwicklungsprojekt (AN)	**Inkrementelle Systementwicklung (AN)** **Komponentenbasierte Systementwicklung (AN)**	**Wartung und Pflege von Systemen (AN)** **Agile Systementwicklung (AN)**
V-Modell-Kern	Projektmanagement Qualitätssicherung Konfigurationsmanagement Problem- und Änderungsmanagement	Messung und Analyse Kaufmännisches Projektmanagement	Projekt genehmigt Projekt definiert Iteration geplant Projekt abgeschlossen
	Systemerstellung	HW-Entwicklung SW-Entwicklung Logistikkonzeption Evaluierung von Fertigprodukten Weiterentwicklung und Migration von Altsystemen Benutzbarkeit und Ergonomie Systemsicherheit	System spezifiziert System entworfen Feinentwurf abgeschlossen Systemelemente realisiert
Schnittstelle zum Auftraggeber	Lieferung und Abnahme (AN) Vertragsschluss (AN)	Lieferung und Abnahme (AG) Vertragsschluss (AG)	Lieferung durchgeführt Abnahme erfolgt Projekt ausgeschrieben Projekt beauftragt Projektfortschritt überprüft



Agile Softwareentwicklung

http://agilemanifesto.org

- Gegenbewegung zu den oft als schwergewichtig und bürokratisch angesehenen Softwareentwicklungsprozessen wie RUP oder V-Modell
- Idee: Schnell eine einsetzbare Software ausliefern und inkrementell (in kurzen Zyklen) in engem Kontakt mit dem Kunden ausbauen
- "Agile Werte" bilden das Fundament der Agilen Softwareentwicklung
  - ⇨ Offenheit und Kommunikation statt Formalismen
- Die Grundprinzipien der agilen SW-Entwicklung (agiles Manifest [Beck,…]) stellen

● Individuen und Interaktion	über	● Prozesse und Werkzeuge
● Lauffähige Software	über	● umfassende Dokumentation
● Zusammenarbeit mit dem Kunden	über	● Vertragsverhandlungen
● Reaktion auf Änderungen	über	● Befolgung eines Plans

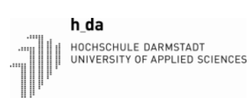
"Obwohl die rechten Punkte ihren Wert haben, bewertet die agile SW-Entw. die linken höher."



Die agile SW-Entwicklung – die Grundidee



- "Der Kunde kennt die wirklichen Anforderungen zum Projektbeginn meist noch nicht komplett! Er fordert Features, die er nicht braucht, und vergisst solche, die benötigt werden!"
- Wie begegnet die agile Methode typische Gefahren für Software-Entwicklungsprojekte:
  - ⇨ Unbenutzbarkeit aufgrund von Programmierfehlern
    - Lösung: Viele und frühe Tests
  - ⇨ Unbenutzbarkeit aufgrund von Fehlentwicklung
    - Lösung: Kunde mit in den Entwicklungsprozess einbeziehen
  - ⇨ Unwartbarkeit
    - Lösung: Viele und frühe Tests
  - ⇨ Zeitplan nicht einhaltbar
    - Lösung: Nur Implementieren, was höchstes Nutzen/Aufwand-Verhältnis hat
  - ⇨ "Featuritis" ("Vielleicht brauchen wir einmal dieses Feature...")
    - Lösung: Lass es!



Die agile SW-Entwicklung – die Grundidee

- verzichte auf einen strikten Anforderungskatalog
  - ⇨ Kundenwünsche, die sich noch während der SW-Entwicklung ergeben, werden einfach noch berücksichtigt
- implementiere nur die im aktuellen Iterationsschritt benötigten Merkmale
- durchlaufe immer wieder in kurzen Zyklen (typ. eine Woche) sämtliche Disziplinen der klassischen Softwareentwicklung
  - ⇨ (Anforderungsanalyse, Design, Implementierung, Test)
- verwende kurze Release-Zyklen und integriere dauernd
  - ⇨ lauffähige Systeme in kurzen Zeitabständen
- erstelle eine gute Dokumentation
  - ⇨ leicht verständlich, vollständig aber nicht ausschweifend

agile Entwicklung ist ein evolutionäres Modell



Agile Vorgehensweisen

Agile SW-Entwicklung ist nicht eine bestimmte, sondern ein Überbegriff für verschiedenen Vorgehensweisen.

- Extreme Programming (K. Beck)
- Adaptive Software Development (J. Highsmith)
- Lean Software Development (M. Poppendieck)
- Crystal (A. Cockburn)
- Scrum (K. Schwaber)
- …

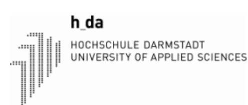
- Wir wollen hier "Extreme Programming" und "Scrum" näher anschauen.



Die agile Vorgehensweise „Extreme Programming (XP)“ - Grundprinzipien

- Prinzipien (Auswahl)
  - ⇨ Sit together - das gesamte Team entwickelt in einem Raum
  - ⇨ Alle Informationen sind jedem zugänglich (z. B. auch auf an den Wänden)
  - ⇨ Kollektives Eigentum - jeder darf jederzeit jeden Quellcode verändern
  - ⇨ Enge Einbeziehung zum Kunden - "Story-Cards" beschreiben

    Anwendungsfälle exemplarisch in Form kurzer Geschichten
  - ⇨ offene und respektvolle Kommunikation - Kritik, Informationen
  - ⇨ Pair-Programming - zwei Programmierer; einer codiert, einer denkt mit (regelmäßig die Rollen tauschen, Vier-Augen-Prinzip)
  - ⇨ Testgesteuerte Programmierung - erst Unit-Tests schreiben, dann Funktionalität programmieren



Die agile Vorgehensweise „Extreme Programming (XP)“ - Grundprinzipien

- Prinzipien (Auswahl)
  - ⇨ Inkrementelles Design, d. h. Design ist nicht von vornherein festgelegt, sondern ständig änderbar / erweiterbar.
  - ⇨ Ständige Integration neu erstellter / veränderter Teile. (Build sollte in 10 Minuten durchführbar sein.)
  - ⇨ Refactoring – systematisches und regelmäßiges Überarbeiten der Ergebnisse: entferne Redundanz, lösche überflüssige Funktionalität, überarbeite das Design!
  - ⇨ Wöchenzyklen: Meeting mit Planung der Arbeit für eine Woche.
  - ⇨ Quartalszyklus: Meeting mit Besprechung der Themen für nächstes Quartal.

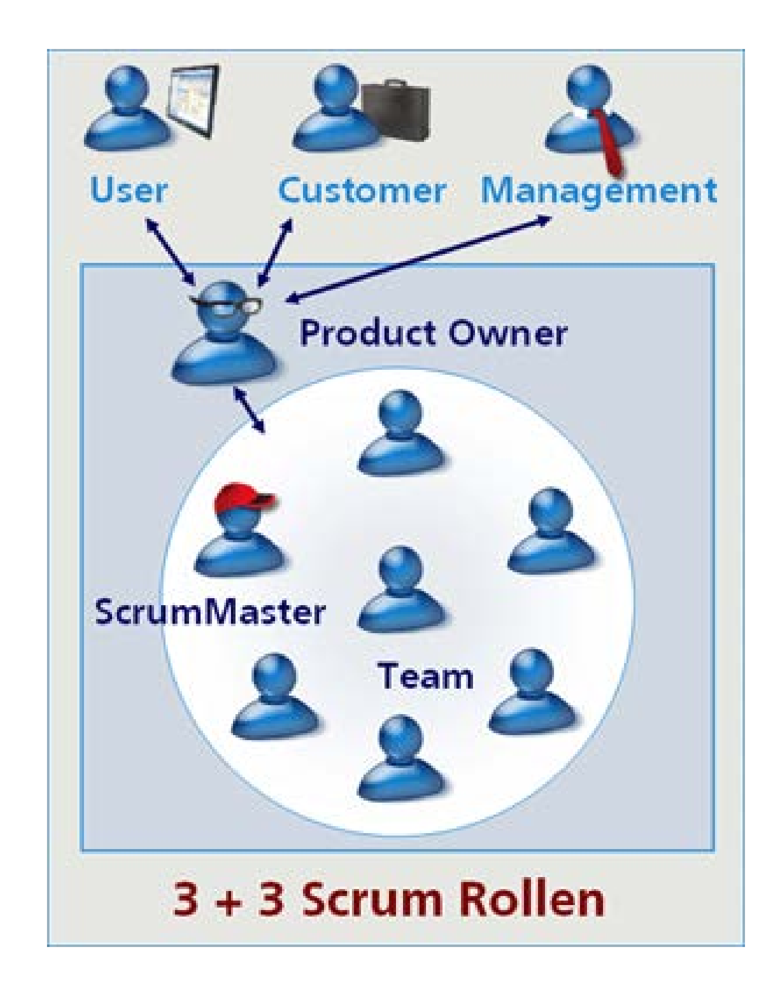


Die agile Vorgehensweise „Scrum“ - Grundprinzipien, Rollen

- Scrum ist ein Prozessmodell für agile Produkt- und Softwareentwicklung.
- Für Scrum sind auch die oben beschriebenen Prinzipien gültig.
- Scrum besteht aus einfachen Arbeitstechniken und Ritualen sowie wenigen Rollen und Artifakten.
- Eine kurze, aber etwas detailliertere Beschreibung finden Sie unter http://www.microtool.de/instep/de/scrum.asp



Die agile Vorgehensweise „Scrum“ - Rollen



Rollen:

Product-Owner:

Nimmt Interessen der Stakeholder wahr,

legt Ziele der Entwicklung fest.

Stellt Finanzierung sicher.

Cross-functional Team:

Organisiert sich eigenständig.

Schätzt jeweils Aufwände für Entwicklung.

ScrumMaster:

Sorgt dafür, dass Team produktiv arbeiten kann.

Ermittelt Verbesserungspotential im Prozess

Optimiert Arbeitsbedingungen.

Quelle:
http://www.microtool.de/instep/de/scrum.asp



Die agile Vorgehensweise „Scrum“ - Arbeitstechniken

Daily Scrum Meeting

24 Stunden

Potenziell lieferfähiges Produkt-inkrement

Sprint Backlog

Product Backlog

2-4 Wochen

Sprint

Scrum Flow

Quelle:
http://www.microtool.de/instep/de/scrum.asp

Aufgabenliste für Sprint erstellt durch Team

Priorisierte Liste von zu liefernden Funktionen (Stories) erstellt durch Product Owner

Sprint Planning Meeting (8 Std.)

- Product Owner präsentiert Product Backlogs mit höchster Priorität.
- Verständnisfragen von Teammitgliedern.
- Team wählt Funktionen für kommenden Sprint.
- Team erzeugt Sprint Backlog.

Daily Scrum Meeting (15 Min.)

- Täglich gleicher Ort u. Zeit
- Alle Teammitglieder
- Nur einer redet
- Jedes Teammitglied:
  - Was habe ich erreicht
  - Was will ich heute erreichen
  - Welche Hindernisse
- ScrumMaster beseitigt Hindernisse

Die agile Vorgehensweise „Scrum“ - Arbeitstechniken

Daily Scrum Meeting

24 Stunden

Sprint Backlog

Product Backlog

2-4 Wochen

Sprint

Potenziell lieferfähiges Produkt-inkrement

Scrum Flow

Quelle: http://www.microtool.de/instep/de/scrum.asp

Aufgabenliste für Sprint erstellt durch Team

Priorisierte Liste von zu liefernden Funktionen (Stories) erstellt durch Product Owner

Sprint Review Meeting (4 Std)
am Ende des Sprints
- Team präsentiert Project Owner neue Funktionalität

Sprint Retrospective Meeting
(15 Min.) nach Review Meeting
- ScrumMaster überlegt mit Team, wie Ablauf des nächsten Sprints verbessert werden kann, so dass Arbeit effizienter wird und mehr Spaß macht

Bewertung: agiles Vorgehen

Vorteile

- Einsatzfähige Produkte für den Auftraggeber in kurzen Abständen
- Integration der Erfahrungen der Anwender in die Entwicklung
- Überschaubare Projektgröße
- Korrigierbare Entwicklungsrichtung
- Keine Ausrichtung auf einen einzigen Endtermin
- Wissensverbreiterung durch Pair Programming
- Weniger Bürokratie / mehr Spaß !?

Nachteile

- Auftraggeber und Team müssen mitspielen (sehr gute Vertrauensbasis zum Auftraggeber erforderlich)
- Erfordert hohe Prozessreife des Teams
- Verzicht auf ein reguläres Anforderungsmanagement erschwert Planung von Kosten und Terminen
- Das Projektmanagement muss frühzeitig kommunizieren und Entscheidungen fällen, andernfalls können geschätzter und realer Aufwand auseinanderlaufen

Agiles Vorgehen ist

⇨ gut geeignet, wenn der Auftraggeber noch nicht sicher ist, was er will!

⇨ *weniger* geeignet, für umfangreiche Projekte mit planbaren Kosten & Terminen!?



Kontrollfragen Vorgehensmodelle

- Wozu sind Vorgehensmodelle wichtig?
- Was beinhaltet ein Vorgehensmodell?
- Was ist ein iterativ- inkrementelles Modell?
- Welche Nachteile hat das Wasserfallmodell?
- Welche Vorteile bietet das V-Modell?
- Welche Klassifikationen von Prototypen gibt es?
- Was halten Sie von der Aussage "In meinem letzten Projekt hat sich das Modell "X" bewährt – so mache ich das jetzt immer"?
- Ist der RUP ein agiles Vorgehens-Modell? Denken Sie an Iterationen, Zykluslängen, Zeitpunkt der Architekturentstehung, Dokumentation)
- Welches Vorgehensmodell haben wir im Praktikum eingesetzt?

Folgendes Kapitel nicht prüfungsrelevant (Kap 11.1)

Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

11.1 Modelle zur Bewertung von Software-Entwickungsprozessen



Lernziel Vorgehensmodelle

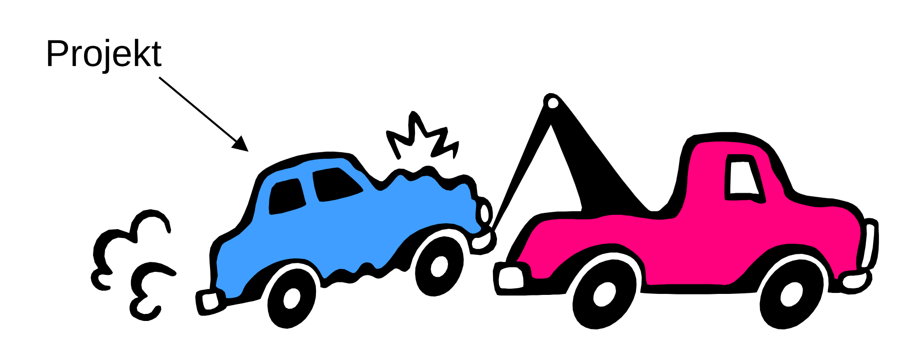
- Sie sollen in diesen Kapitel lernen,
  - ⇨ wie man im Modell SPICE vorgeht, um die Güte von Software-Entwickungsprozessen zu bewerten
  - ⇨ was die Unterschiede zu CMMI sind (CMMI ist ein anderes Model zur Bewertung von SW-Entwicklungsprozessen)
  - ⇨ was die Unterschiede zu V-Modell-XT bzw. RUP sind

Anschließend wissen Sie wie man die
Qualität eines Prozesses messen kann!



Einführung

Jedem hier ist klar::



Seit dem es Computer gibt

- scheitern SW-Projekte,
- zugesagte Termine werden nicht eingehalten,
- kalkulierte Kosten werden überschritten,
- die Qualität der SW ist viel geringer als erwartet.

=> Wir brauchen einen klar strukturierten SW-Entwicklungsprozess.

=> Wir müssen bewerten können, wie gut man die durchzuführenden Aufgaben mit dem eingesetzten SW-Entwicklungsprozess lösen kann.



Prozess-Bewertungs-Modell, SW-Entwicklungsprozess-Modelle und Projekte

Prozess-Bewertungs-Modelle

SPICE,
CMMI,
...

Messen die Qualität des SW-Entwicklungsprozesses in Organisationen (nicht die Qualität der Software!!) Prinzipien, die beachtet werden müssen.

Prinzipien sollten beachtet werden

Assessment / Appraisal

SW-Entwicklungsprozess-Modelle

V-Modell XT,
RUP,
...

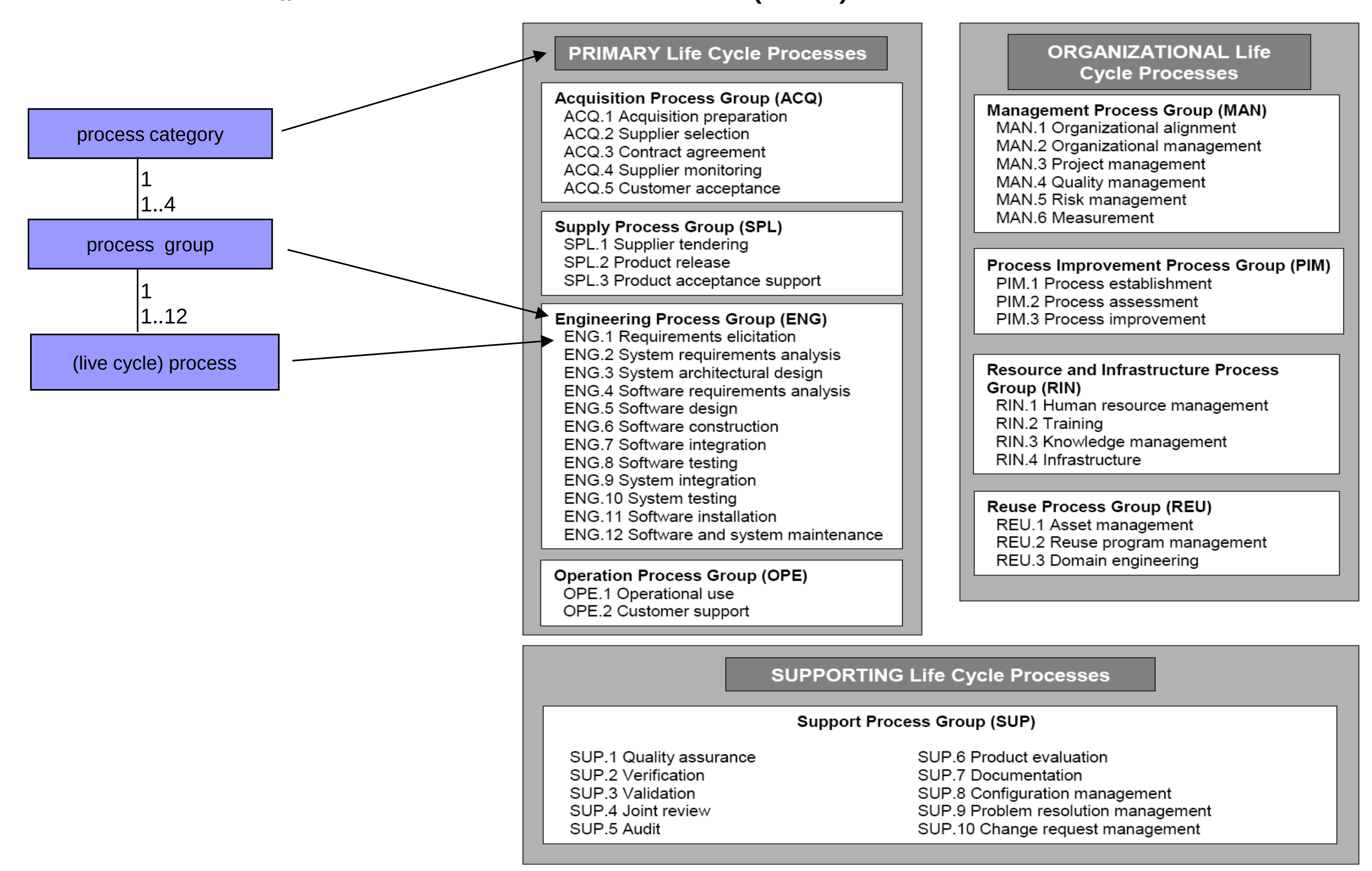
Durch „Tailoring“ wird der SW-Entwicklungsprozess an die Erfordernisse des Projektes angepasst (Tools, Templates, Methoden, Rollen,..)

Wird verwendet von

Projekte in der Organisation

Projekte verwenden einen projektspezifischen SW-Entwicklungsprozess (Planung, Abwicklung des Projektes...)

Was ist SPICE? – Das „Process Reference Model“ (PRM) - ISO/IEC 15504-5



SPICE – „Process Assessment Model“ (PAM)

Die Qualität eines Prozesses (z. B. ENG.1) ist bewerted mit einem Capability Level:

SPICE - Capability Level 0 und 1

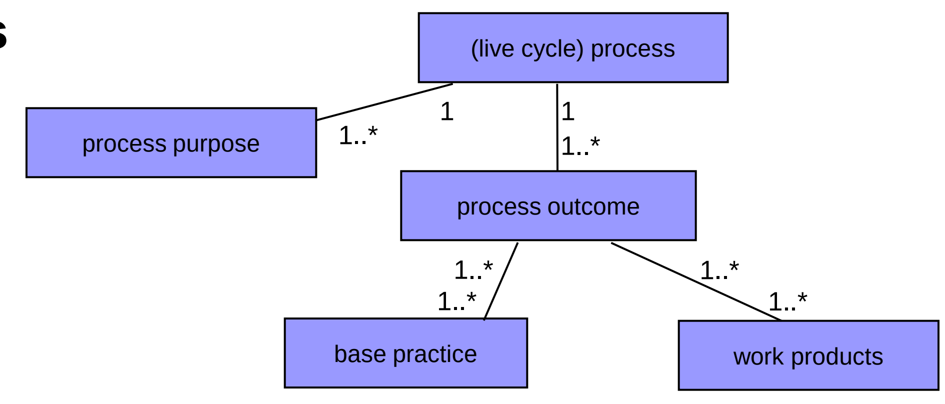
Wie sind die „Capability-Levels“ (der „Live Cycle Processe“) definiert?

Level 0: Incomplete (unvollständiger) **Process**

- Der Prozess ist nicht implementiert.

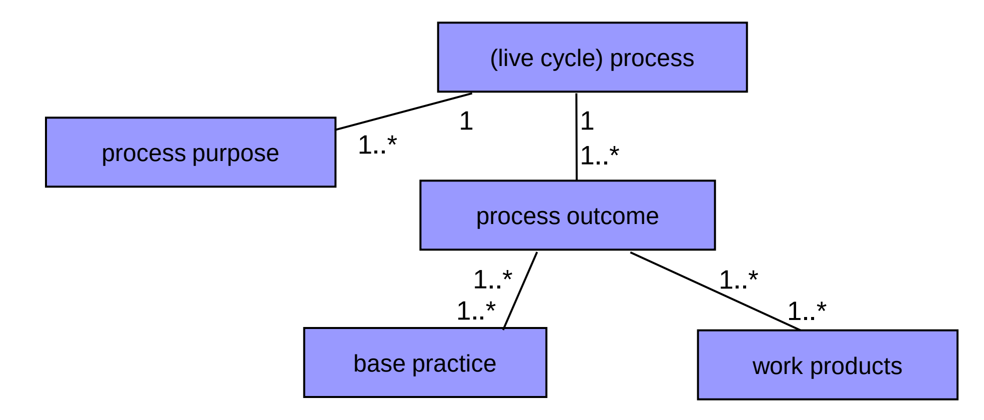
Level 1: <u>Performed</u> (durchgeführter) **Process**

- Level 1 ist erreicht, wenn der implementierte Prozess den Zweck (“*<u>Process Purpose</u>*”) des Prozesses erfüllt. Zur Bewertung wird geprüft ob
- die “*<u>Process outcomes</u>*” erzielt werden,
- die “*<u>base practices</u>*” implementiert sind,
- die “*<u>input and output work products</u>*” existieren .



SPICE - Beispiel ENG.1 (aus ISO/IEC 15504-5)

Process ID:
ENG.1
Process name:
Requirements elicitation
Process purpose:
The purpose of the requirements elicitation process is to gather, process, and track evolving customer needs and requirements
...
Process outcomes:
As a result of successful implementation of requirements elicitation process
1. continuing communication with the customer is established;
2. agreed customer requirements are defined and baselined;
3. a change mechanism is established;
4. a mechanism is established for continuous monitoring of customer needs;
...
Base practices:
Obtain customer requirements and requests (Outcome: 1, 4)
Agree on requirements (Outcome: 2)
Establish customer requirements baseline (Outcome: 2, 3)
...
Input work products
Commitment / agreement (Outcome: 2)
Change request (Outcome: 3)
...
Output work products
Customer requirements (Outcome: 2, 3)
Communication record (Outcome: 1, 4)
Change control record (Outcome: 3, 4)
...





SPICE - Capability Level 2, „Process Attributes” and “Indicators”

Level 2: Managed Process

- Der “Performed (durchgeführte) Process” (Level 1)
  - ⇨ wird zusätzlich geplant, verfolgt und angepasst und
  - ⇨ Die Arbeitsprodukte sind adequat implementiert, stehen unter Konfigurationsmanagement, werden qualitätsgesichert, gemanaged und fortgeschrieben

Für die Bewertung eines Processes genügt keine informelle Beschreibung

=> “Process Attributes” und “Indicators”

- Zu ***Capability Level*** 2 oder höher existieren jeweils 2
  - ⇨ ***“Process Attributes”***
  - ⇨ Für jedes “Process Attribute” sind “Indicators” definiert:
    - ***“Generic Practices”***
    - ***“Generic Resources”***
    - ***“Generic Work Products”***

  Diese “Indicators” helfen dem Assessor den “Capability-Level” des Prozesses zu bestimmen.

SPICE - Attributes, Generic Practices, Generic Resources, Generic Work Products für Level 2: Managed Process (aus ISO/IEC 15504-5)

Process Attribute 2.1: Performance Management
is a measure of the extent to which the performance (Durchführung) of the process is managed.
...

- **Generic Practices:**
  Identify the objectives (Ziele) for the performance of the process.
  Plan and monitor the performance of the process.
  Adjust the performance of the process (if planned results are not achieved).
  ...
- **Generic Resources**:
  project planning, management tools and control tools,
  communication mechanisms,
  ...
- **Generic Work Products**:
  Documentation of the objectives,
  milestones and timetable,
  results and status of the process,
  ...

Process Attribute 2.2: Work Product Management
is a measure of the extent to which the work products produced by the process are appropriately managed. (Configuration management)
...



SPICE - Capability Level 3

Level 3: Established Process

- Der Live-Cycle-Process existiert als ein organisationseinheitlich festgelegter *Standard-Prozess*
- Ein Projekt verwendet eine angepasste ("*tailored*") Version dieses Standard-Prozesses, *=> definierter Prozess*.
- Während der Prozessausführung
  - Identifikation von Schwachstellen
  - Verbesserung des Standardprozesses

- Der Established Process hat auch wieder 2 Process Attributes, für die die Generic Practices, Generic Resources, Generic Work Products nachgeprüft werden

SPICE - Capability Level 4

Level 4: Predictable Process

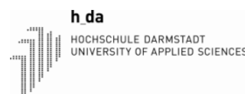
- In jedem Projekt werden “Schlüsseldaten” gesammelt wie z. B.:
  - Anzahl der im Integrationstest gefundenen Fehler pro Zeile,
  - statistische Daten über den Grund der Fehler, ...
- Es können Schlüsseldaten von unterschiedlichen Projekten verglichen werden (wegen Einführung Standardprozess).
- Es können statistische Auswertungen über den archivierten Schlüsseldaten gemacht werden.
  - ⇨ Wir können Konsequenzen von Entscheidungen voraus sagen.
  - ⇨ Wir können vor dem Eintreten negativer Auswirkungen reagieren.
  - ⇨ Das Management kann das Projekt objektiv bewerten.

SPICE - Capability Level 5

Level 5: Optimizing Process

- Quantitative Prozessziele werden gesetzt und die Einhaltung verfolgt.
- Prozesse werden fortlaufend verbessert
- Innovative Ansätze werden erpobt und über erhobene Schlüsseldaten bewertet.

- Bemerkung: Wegen des hohen Aufwands wird normalerweise nur das Erreichen von Capability Level 2 oder 3 angestrebt.



SPICE – Bestimmung des Capability Level (Fähigkeitsgrad) eines Prozesses

Wir Bewerten das Erfülltsein von

- Prozess-Attributen bzw. das
- Durchführen von generischen Praktiken,
- Verfügbarsein von generischen Resourcen und
- Vorhandensein von Arbeitsergebnissen

Wann ist ein bestimmter Capability Level erreicht?

Von	Multiplizität	Beziehung	Multiplizität	Nach
process category	1		1..4	process group
process group	1		1..12	(live cycle) process
capability level	1	has	0..*	(live cycle) process
capability level	1		0..2	process attribute
(live cycle) process	1		1..*	process purpose
(live cycle) process	1		1..*	process outcome
process attribute	1		1..*	generic practice
process attribute	1		1..*	generic resource
process attribute	1		1..*	generic work product
process outcome	1..*		1..*	base practice
process outcome	1..*		1..*	work products



SPICE - Evaluation der Prozess Attribute und Bestimmung des Capability Levels des Prozesses

Bewertung des Erfülltseins eines Prozess-Attributes:

- F: Fully achieved.
- L: Largely achieved
- P: Partially achieved
- N: Not achieved

Um F zu erreichen

- ⇨ müssen nicht unbedingt alle Praktiken implementiert sein,
- ⇨ müssen nicht unbedingt alle geforderten Resourcen verfügbar sein,
- ⇨ müssen nicht unbedingt alle Arbeitsergebnisse erstellt werden.
- ⇨ Es hängt vom speziellen Umfeld und der Meinung des Assessorenteams ab, was unbedingt vorhanden sein muss.

- Ein Level ist erreicht, wenn

  alle Prozess-Attribute

  - ⇨ des Levels mindestens “largely achieved” sind und
  - ⇨ die Prozess-Attribute der darunterliegenden Levels “fully achieved” sind.



Was ist CMMI?

- Die Zielsetzung vom CMMI ist die gleiche wie die Zielsetzung von SPICE:
  - ⇨ Bewertung des SW-Entwicklungsprozesses (in CMMI zus. System-Entw.-Process)
- CMMI bietet 2 Methoden zur Bewertung an:
  - ⇨ Die ***kontinuierliche Darstellung*** (***continuous representation)*** ist SPICE sehr ähnlich:
    - Bewertet einzelne “***Process Areas”*** (~“Live Cycle Processes" in SPICE) mit einem ***“Capability Level”***
    - wie im “***Target Profile”*** definiert (~“Process Assessment Modell" in SPICE)
  - ⇨ Die ***stufenförmige Darstellung (staged representation)***
    - bewertet die Organisation mit einem **Reifegrad *(maturity level)***.

      (Im Normungsgremium von SPICE wurde inzwischen auch eine stufenförmiger Darstellung erarbeitet.)



CMMI Continuous Representation

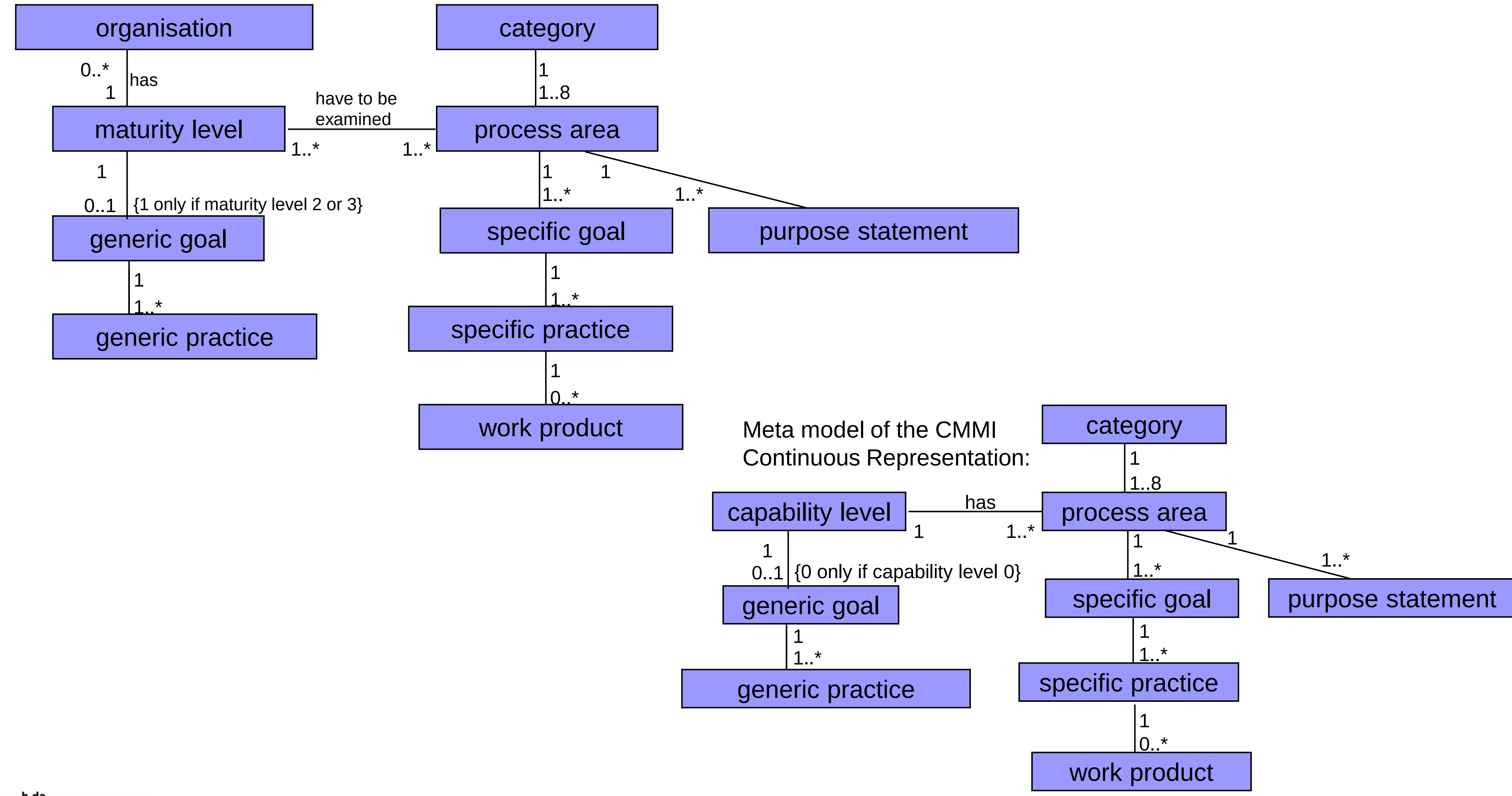
category
1
1..8
capability level
has
1
0..*
process area
1
0..1
{0 only if capability level 0}
1
1..*
1
1..*
generic goal
specific goal
purpose statement
1
1..*
1
1..*
generic practice
specific practice
1
0..*
work product

SPICE meta model

process category
1
1..4
process group
1
1..12
capability level
1
has
0..*
(live cycle) process
1
0..2
1
1..*
1
1..*
process attribute
process purpose
process outcome
1
1..*
1
1..*
1
1..*
1..*
1..*
1..*
1..*
generic practice
generic resource
generic work product
base practice
work products



CMMI Staged Representation



V-Modell XT

project type
process module group
project execution strategy
process module
product/activity group
decision gate
work products
role
activity
activity step
optional
mandatory
{ordered as workflow}
contribute
responsible
is dependant on
1..*
1
1..4
1..9

SPICE meta model

process category
process group
(live cycle) process
capability level
has
process attribute
process purpose
process outcome
generic practice
generic resource
generic work product
base practice
work products
1..12
0..2
0..*



Impliziert die Verwendung des V-Modells XT eine gute Bewertung in SPICE oder CMMI?

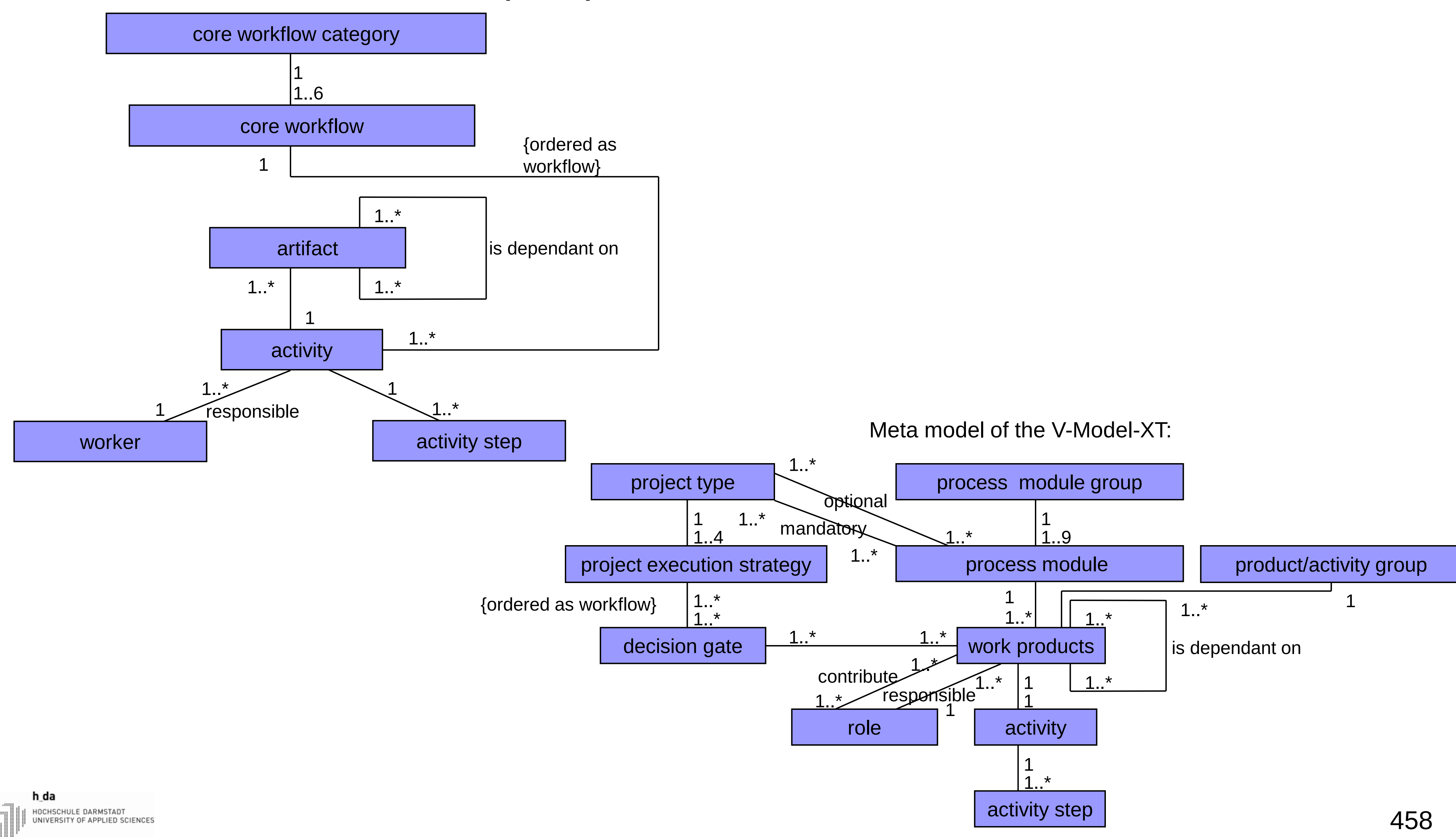
- In der Beschreibung des V-Modells XT wird gezeigt, dass die einzelnen spezifischen und generischen Ziele von Level 2 und 3 des CMMI auf Vorgaben im V-Modell XT abgebildet werden können.
- Das heißt nicht, dass bei Verwendung von V-Modell XT automatisch CMMI-Level 3 erreicht ist.
- CMMI ist ein Modell zur Überprüfung des Fähigkeitgrads bzw. Reifegrades eines Entwicklungsprozesses.
  - ⇨ Es muss unabhängig von der Verwendung eines Vorgehensmodells geprüft werden inwieweit die spezifischen und generischen Ziele bei der Umsetzung des Prozesses wirklich erreicht sind.

=> Durch die Verwendung des V-Modells XT ist eine gute Basis gelegt für eine gute Bewertung in CMMI, aber erst nach dem Appraisal-(Begutachtungs-)Prozess kann entschieden werden, welcher Level erreicht ist.

- Die gleiche Argumentation gilt für SPICE



Rational Unified Process (RUP)



Impliziert die Verwendung des RUP eine gute Bewertung in SPICE oder CMMI?

- Wir haben hier die gleichen Argumente wie beim V-Modell XT:

⇒ Durch die Verwendung des RUP ist eine gute Basis gelegt für eine gute Bewertung in SPICE oder CMMI, aber erst nach dem Appraisal-Prozess kann entschieden werden, welcher Level erreicht ist.



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

12. Technisches Management



Quellenhinweis:
Einige Folien dieses Kapitels entstammen Präsentationen von Prof. A. del Pino

Änderungsmanagement (Change Management)

- Anforderungen ändern sich im Lauf der Zeit
  - ⇨ neue Anforderungen kommen hinzu
  - ⇨ bestehende Anforderungen werden modifiziert
  - ⇨ Die Prioritäten bestehender Anforderungen ändern sich
- "Change Management": **Systematischer Umgang mit Änderungsanträgen**
  1. Analyse der Veränderungen gegenüber geplanten Zeiten, Kosten und Risiken
  2. Analyse der Auswirkung dieser Veränderung auf andere Anforderungen
  3. Freigabe der Änderung und Einarbeitung in das Pflichtenheft, Designdokumente,...

  - ⇨ Achtung! Die *Analyse* jedes Change Requests erfordert bereits Ressourcen
  - ⇨ Für die Umsetzung müssen oft Kosten nachverhandelt werden

Change Management ist eine Gratwanderung zwischen Kundenzufriedenheit und Einhaltung des Projektplans



Risikomanagement (I)

- In jedem Projekt bestehen Risiken
  - ⇨ Mitarbeiter können kündigen oder das Projekt verlassen
  - ⇨ Getroffene Annahmen stellen sich als falsch heraus
    - z.B. Arbeitspaket X ist in 3 Tagen fertig
  - ⇨ Zulieferungen bleiben aus oder verspäten sich
    - z.B. Technologie X ist bis zum Termin Y verfügbar
  - ⇨ ...
- *Ein Risiko ist der Zusammenhang zwischen einer Ursache, die mit einer gewissen Wahrscheinlichkeit eintreten kann, und einer in der Regel negativen Auswirkung, bei der ein Verlust entsteht*
  - ⇨ Die negative Auswirkung muss nicht zwangsläufig eintreten – man kann auch Glück haben
  - ⇨ Der Verlust kann zum Beispiel eine zeitliche Verzögerung, zusätzliche Kosten oder eine Verschlechterung der Qualität sein

Wie geht man mit Risiken um?



Risikomanagement (II)

- Ein *Risikomanagement-Prozess* ist das systematische Zusammenspiel von Prozessbausteinen mit dem Ziel, Risiken letztlich akzeptierbar zu machen
  1. Risikoidentifikation
     - Nicht alle Risiken sind unmittelbar bekannt. Sie müssen proaktiv gesucht und entdeckt werden
     - Das Ergebnis ist ein Dokument, das für jedes Risiko neben seiner Beschreibung, auch die Wahrscheinlichkeit seines Auftretens und den möglichen Verlust identifiziert
  2. Risikoanalyse
     - In der Risikoanalyse werden die erkannten Risiken nach wohldefinierten Kriterien analysiert, und nach ihrer relativen Bedeutung für das Projekt priorisiert
     - Das Ergebnis der Risikoanalyse ist eine priorisierte Risikoliste
  3. Risikoplanung
     - Auf der Basis der priorisierten Risikoliste werden in einem Aktionsplan Szenarien entwickelt, und die Alternativen zur Auflösung von Risiken beschrieben
     - Es werden Schwellwerte festgelegt, wann der Aktionsplan zum Einsatz kommen soll
     - Das Ergebnis der Risikoplanung ist ein Aktionsplan



Risikomanagement (III)

- Der Risikomanagement-Prozess (fortgesetzt)

  4. Risikoverfolgung
     - Hierbei wird überprüft, ob während des Projektverlaufs bestimmte Metriken die im Aktionsplan festgelegten Schwellwerte überschreiten, und gegebenenfalls werden bestimmte Abläufe des Aktionsplans ausgelöst
  5. Risikoauflösung
     - Die Risikoauflösung ist die Reaktion auf einen Auslöser, wobei dann bestimmte Teile des Aktionsplans durchgeführt werden

⇨ Risiken in einem Projekt sind normal und unvermeidbar

⇨ Als Entwickler sind Sie vor allem an der Risikoidentifikation und Risikoanalyse beteiligt

⇨ Risikomanagement erlaubt den bewussten Umgang mit diesen Risiken und verhindert böse Überraschungen

⇨ Evtl. auch Einsatz von Prototypen zur Risikobewertung



Konfigurationsmanagement (I)

Quelle: IEEE Standard Glossary of Software Engineering Terminology

- A system is a collection of components (=Systemteile) organized to accomplish a specific function or set of functions
  - ⇨ Hardware-Komponenten
  - ⇨ Software-Komponenten
  - ⇨ Firmware-Komponenten
  - ⇨ ...
- Die Konfiguration eines Systems beschreibt die Komponenten, die jeweils in einer bestimmten Version vorliegen und mit einer wohl definierten Prozedur zu dem System zusammengesetzt werden können
- Während der Entwicklung eines Systems entstehen viele Konfigurationen
  - ⇨ Jedes Release entspricht (mindestens) einer Konfiguration
  - ⇨ Jedes Testszenario entspricht einer Konfiguration
  - ⇨ Jeder Zwischenstand in der Integration entspricht einer Konfiguration
  - ⇨ ...



Konfigurationsmanagement (II)

- Beim *Konfigurationsmanagement* geht es darum, die vielen entstehenden Konfigurationen zu verwalten
  - ⇨ systematische und nachvollziehbare Durchführung von Änderungen
    - Was wurde gegenüber der letzten Konfiguration geändert?
    - In welchen Konfigurationen wird diese Version der Komponente noch verwendet?
  - ⇨ Wiederherstellung von "alten" Konfigurationen
    - wenn z.B. ein Fehler zu einem Release gemeldet wird
- Konfigurationsmanagement-Tools
  - ⇨ verwalten sehr viele Dateien und zusätzliche Datenstrukturen darauf
  - ⇨ verwalten Versionen von Quellcode (bekannt als CheckIn und CheckOut)
  - ⇨ bieten Funktionen zum Suchen, Verzweigen, und Zusammenführen von Dateien
  - ⇨ Beispiele (Open Source): Subversion (SVN), Concurrent Version System (CVS)

Eigentlich müssten auch die Hardware, das Betriebssystem, die Entwicklungsumgebung uvm. mitverwaltet werden, damit eine Konfiguration nach vielen Jahren tatsächlich wiederherstellbar ist



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

13. Zusammenfassung



Themen in der Veranstaltung

- Software Engineering im Projekt
  - ⇨ Anforderungsanalyse (Spezifikation)
  - ⇨ Abnahme und Systemtests
  - ⇨ Architektur
  - ⇨ Design
  - ⇨ Implementierung
  - ⇨ Test
- Querschnittsthemen
  - ⇨ Qualitätsmanagement
  - ⇨ Software-Metriken
  - ⇨ Vorgehens- und Prozessmodelle
  - ⇨ Technisches Management
  - ⇨ (Prozessorientiertes Qualitätsmgmt (CMMI, SPICE, Assessments) -leider nicht geschafft)



Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

13.1 Ergebnisse im Praktikum



Inhalt des Praktikums

- Der CocktailPro war kein wirklich großes Projekt....
  - ⇨ (relativ) geringer Umfang, kurze Laufzeit, kleines Team
  - ⇨ keine Wartung
  - ⇨ keine Change Requests
  - ⇨ wenige Schnittstellen nach außen (z.B. Zulieferung von Fremdcode)
  - ⇨ keine Ausnahme- und Fehlerbehandlung
- ....aber, es wurden dennoch viele Themen behandelt
  - ⇨ OOA, OOD, Modellierung (Use Cases, Klassendiagramme, Sequenzdiagramme)
  - ⇨ Implementierung (Guidelines, Objektorientierung)
  - ⇨ Beobachter-Muster
  - ⇨ Integration (Zulieferung Rezeptbuch / verteilte Entwicklung)
  - ⇨ Test (Blackbox, Whitebox, C++Unit)
  - ⇨ Review
  - ⇨ Metriken
  - ⇨ CASE-Tool, Round-Trip-Engineering



Wie gut ist Ihr Entwurf?

- Analysieren Sie bitte folgende Change Requests:
  1. es wird ein neues Rezeptbuch mit neuen Rezepten zugeliefert
  2. das User-Interface soll in Deutsch und Englisch verfügbar sein
  3. es wird in den Rezepten ein neuer Verarbeitungsschritt "Rühren" eingeführt
  4. es soll bei einem Füllstand von 200g ein Warnungs-Piepton ausgegeben werden
  5. die gleiche Zutat kann in mehreren Dosierern sein (es soll automatisch umgeschaltet werden, wenn ein Dosierer leer ist)
  6. die Cocktails sollen schneller gemischt werden, indem die Zutaten parallel abgewogen werden. Dazu erhält jeder Dosierer eine eigene Messvorrichtung.

⇨ In welchen Klassen und Methoden müssen Sie Änderungen machen?
⇨ Ist die Änderung eher ein Hack oder passt sie ins Design?
⇨ Könnte die Änderungen auch leicht jemand anderes machen?
⇨ Wie viele Stunden brauchen Sie zur Umsetzung?
⇨ Könnten Sie diese Änderungen auch im Code Ihres Kollegen machen?



Lessons Learned? (I)

- Haben Sie den Wert der Modelle erkannt?
  - ⇨ die Abstraktion hilft beim Bewahren des Überblicks
  - ⇨ die Darstellung erleichtert das Verständnis
  - ⇨ In den Modellen werden Entscheidungen getroffen und wichtige Vorgaben gemacht
  - ⇨ die Implementierung wird geradlinig und einfacher
    (oder auch nicht, falls die Modelle nicht adäquat sind)

- ...aber, falls der Detaillierungsgrad nicht stimmt, klappt es nicht
  - ⇨ z.B. Implementierung trotz unvollständiger Klassen- und Sequenzdiagramme führt zu häufigen Schnittstellenänderungen
  - ⇨ ...

Lessons Learned? (II)

- Haben Sie den Wert von Metriken erkannt?
  - ⇨ schnelle und frühe Analyse der Ergebnisse
  - ⇨ wertvolle Hinweise auf potenzielle Schwachstellen
  - ⇨ Anregungen zur Verbesserung des persönlichen Entwicklungsstils
- ...aber, auch die Gefahren?
  - ⇨ übertrieben gute Ergebnisse für halbfertigen Code (keine Fehlerbehandlung.)
  - ⇨ die Erfassung und Auswertung zu vieler Metriken verursacht viel Arbeit
- Haben Sie jetzt besser verstanden was Design ist?
  - ⇨ Geheimnisprinzip, schwache Kopplung und starker Zusammenhalt
  - ⇨ Vererbung und Überladung als praktische Techniken
  - ⇨ Verwendung von Klassen und Sichtbarkeit
- Haben Sie den Nutzen von Design Patterns jetzt besser verstanden?
  - ⇨ Observer als praktisches Pattern
  - ⇨ Tipp: Das (einfache) Singleton-Pattern hätte einiges stark vereinfacht...

Hochschule Darmstadt
Fachbereich Informatik

Software Engineering

13.2 Gesamtergebnis



Häufige Mängel in Software-Projekten... sind vermeidbar

How the customer explained it

How the Project Leader understood it

How the Analyst designed it

How the Programmer wrote it

How the Business Consultant described it

How the project was documented

What operations installed

How the customer was billed

How it was supported

What the customer really needed



Häufige Mängel in Software-Projekten… sind vermeidbar

Anf.-analyse
Pflichtenheft
Lastenheft

Architektur, Design,
Implementierung

Projekt-
Management

Reviews,
Vorgehens-
Modelle,
Metriken

How the customer explained it

How the Project Leader understood it

How the Analyst designed it

How the Programmer wrote it

How the Business Consultant described it

How the project was documented

What operations installed

How the customer was billed

How it was supported

What the customer really needed

Anforderungs-
Analyse
Pflichtenheft
Lastenheft

Model-
lierung,
Dokumen-
tation

Integration,
Test,
Abnahme

Projekt-
Management



Kritische Bemerkungen zum Schluss

- Modelle und Prozesse sind wichtige Instrumente auf dem Weg zur Software
  - ⇨ aber die Software ist das Ziel, nicht das Modell oder der Prozess!
  - ⇨ Modelle sind sinnvoll, solange sie Aufwand reduzieren (auch in der Wartung!) – das Modell darf aber kein Selbstzweck werden !
  - ⇨ In der Modellierung geht es nicht darum, die neuesten UML-Features zu verwenden – Sie dokumentieren für Ihre Kollegen und für die Wartungsphase! Verwenden Sie einfach eine Notiz im Modell um Hinweise zu geben
  - ⇨ Dokumentieren Sie auch Zusatzinformationen wie
    - Zwischenergebnisse,
    - grundsätzliche Ideen und Lösungsansätze,
    - nicht so leicht ersichtliche Randbedingungen und
    - die Gründe bei einer Entscheidung mit Alternativen
  - ⇨ Ein Prozess soll helfen! Wenn er mehr behindert als nützt, ist er der falsche (für dieses Projekt)



Kontrollfragen zum Software Engineering

- Wie stellen Sie in einem großen Projekt sicher,
  - ⇨ dass eine Person nicht alles wissen muss?
  - ⇨ dass Qualitätsauflagen (intern oder vom Auftraggeber) erfüllt werden?
  - ⇨ dass die Wartungsverpflichtung berücksichtigt wird?
  - ⇨ dass Ihr Kunde auch das kriegt was er will?
  - ⇨ dass Sie auch in Teams an verschiedenen Standorten arbeiten können?
  - ⇨ dass die Ergebnisse der Teams auch zusammen passen?
  - ⇨ dass die Software stabil läuft?
  - ⇨ dass die Entwickler nicht alles neu erfinden?

Sie sollten jetzt in einem großen Projekt mitarbeiten können!



ENDE



- Ergänzungsfolien



Operationen der Schnittstellen-Klasse zu Rezeptbuch

```
class SKRezeptbuch
{
public:
void  SKRezeptbuch();

int      GetAnzahlRezepte();

int      GetAnzahlRezeptschritte (int NrRezept);

string GetNameRezept (int NrRezept);

string GetZutat (int NrRezept, int NrRezeptschritt);

float   GetMenge (int NrRezept, int NrRezeptschritt);  …
}
```



Operationen der Realisierungs-Klasse in Komponente KRezeptbuch

```
class KompRezeptbuch::public SKRezeptbuch
{
public:
void KompRezeptbuch()
        { Rezeptbuch dasRezeptbuch };
int GetAnzahlRezepte()
        { return dasRezeptbuch.GetAnzahlRezepte() };
int GetAnzahlRezeptschritte (int NrRezept)
        { return dasRezeptbuch.getRezept(NrRezept)->getAnzahlRezeptschritte() };
string GetNameRezept (int NrRezept)
        { return dasRezeptbuch.getRezept(NrRezept)->getName() };
string GetZutat (int NrRezept, int NrRezeptschritt)
        {return dasRezeptbuch.getRezept(NrRezept)
        ->getRezeptschritt (NrRezeptschritt)->getM_Zutat() };
float GetMenge (int NrRezept, int NrRezeptschritt)
        { ...->GetM_Menge};    …
}
```



- Nicht aktuelle Folien:

Cocktail-Mischmaschine



marães in Portugal (neben dem slowenischen Maribor Kulturhauptstadt Europas), Santiago de Chile, Hongkong, Orlando im US-Staat Florida sowie das australische Darwin. Bei den Ländern ist das Top-Ziel eine Überraschung: Uganda. Es folgen Birma (Myanmar/Burma) und die Ukraine.

STRICK-SAKKO: „Die Strickjacke ist das neue Sakko", behauptet die Modefirma Marc O'Polo, und in der Tat: Europaweit sieht man junge modebewusste Männer mit sogenannten Cardigans, die jedoch nicht opahaft, sondern schick und ein bisschen sportlich wirken. In der Trendsetter-Metropole Istanbul zum Beispiel tragen auf der beliebten Flaniermeile Istiklal Caddesi viele Teenager und Twens die oft dicken, wollig-wärmenden Strickjacken – bevorzugt mit Knöpfen (nicht mit Reißverschlüssen). Oft nehmen die Strickjacken tatsächlich Bezug auf Sakkos und sind als Zweireiher gestaltet. Der „Ralph Lauren Styleguide" schreibt: „Lassen Sie ruhig einmal das Sakko weg und kreieren Sie stattdessen mit Strickjacke, Hemd und Krawatte einen lässigen Look."

MIX-MASCHINE: Kaffee, Suppen oder Nudelgerichte lassen sich auf die Schnelle aus Automaten ziehen. Auch Cocktail-Fans können inzwischen die Maschine für sich arbeiten lassen. Zwei Wirtschaftsingenieure von der Bremer Uni haben einen Automaten entwickelt, der neun verschiedene Cocktails mixt. Die Rezepte für „Sex on the Beach" oder „Screwdriver" sind auf kleinen Chipkarten gespeichert, die die Benutzer vor ein Lesegerät halten müssen. Aus mehreren Tanks mischt der Automat dann das richtige Verhältnis von Schnaps, Säften und Sirup zusammen. Echte Barkeeper wenden sich mit Grausen, doch als Mietgerät für Partys kommt die Maschine an.



Cocktail-Kasten: Kolja Schmidt (links) und Marco Lewandowski mit ihrem Getränkeautomaten Foto: Jaspersen